

THE APPLICATION OF OBJECT-ORIENTED COMPUTING IN THE
DEVELOPMENT OF DESIGN SYSTEMS FOR AUDITORIA

By

GANAPATHY MAHALINGAM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1995

Copyright 1995

by

Ganapathy Mahalingam

This work is dedicated to my parents. They did not have the benefit of a higher education, but they made sure that their children did not miss the opportunity to have one.

ACKNOWLEDGEMENTS

A work of this nature is the culmination of a long, arduous, journey. There are many people to thank for showing me the way. These people have helped me stay on the path and stopped me from going astray.

First, I would like to thank my parents, who wholeheartedly supported me in the pursuit of an architectural education, even when they did not understand its idiosyncrasies.

I would like to thank Rabindra Mukerjee for introducing me to the field of computer-aided design in architecture and for giving me the opportunity to teach at Iowa State University in my formative years.

I would like to thank Dr. Earl Starnes for providing constant intellectual stimulus during my doctoral studies and for being a critical listener when I rambled on with my ideas.

I would like to thank Gary Siebein for exposing me to the intriguing field of architectural acoustics and providing me with the research data needed for part of my dissertation.

I would like to thank Dr. Justin Graver for teaching me more than I wanted to know about object-oriented computing.

I would like to thank my fellow doctoral students, who acted as sounding boards for my ideas and asked the most frustrating questions.

I would like to thank the numerous members of the ACADIA family with whom I have not interacted directly, but whose work has constantly been shaping mine.

I would like to thank my wife, Gayatri, who came into my life during the last stages of writing my dissertation and goaded me to complete it.

Last, but not least, I would like to thank Dr. John Alexander, my mentor, for forcing me to graduate from being a user of computer-aided design systems to a developer of such systems and providing the resources necessary to accomplish this work.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
LIST OF FIGURES	xi
ABSTRACT	xv
CHAPTERS	
1 INTRODUCTION	1
Field of Inquiry	1
Computable Processes and Architectural Design	2
The Common Ground	7
Organization of the Dissertation	9
Origins of Object-oriented Computing	10
Key Concepts of Object-oriented Computing	11
The Object as a Computer Abstraction	12
Encapsulation	14
Information Hiding	16
Computation as Communication	18
Polymorphism	19
Dynamic Functionality	20
Classes and Inheritance	21
Composite Objects	24
The Paradigm Shift	24
Building Blocks	25
Problem Decomposition	26
Top-down Approach versus Unlimited Formalization	27
Encapsulation versus Data Independence	30
Information Hiding	33
Static Typing and Dynamic Binding	34
Serial Computation versus Parallel Computation	36
Classes and Inheritance	37
Analysis, Design and Implementation	38

	The Transition to Object-oriented Computing	39
	Computable Models of Architectural Design	41
	Computable Models for Making Architectural Representations	42
	Computable Models of Architectural Design Decision Making	45
	First-order Computer-based Design Systems in Architecture	54
	Existing Systems	56
	Methodology of the Dissertation	57
2	METHODS	59
	Acoustic Sculpting	59
	The Method of Acoustic Sculpting	61
	Acoustical Parameters	62
	Subjective Perceptions Related to Acoustical Parameters	70
	Selection of Acoustical Parameters	75
	The Generative System	77
	The Implemented Object-oriented Design Systems	84
3	RESULTS	90
	The Computer Model of the Auditorium	90
	Instance Variables	90
	Methods	97
	Results Achieved Using the Design Systems	105
	Validation of the Computer Model of the Auditorium	120
4	DISCUSSION	121
	A New Computable Model of Architectural Design	121
	Architectural Entities as Computational Objects	123
	Interaction of Architectural Computational Objects	127
	Benefits of Object-oriented Design Systems in Architecture	132
	The Object-oriented Perspective	133
	Abstraction	133
	Fuzzy Definitions	134
	Context Sensitive Design Decision Making	135
	Multiple Representations	135
	The Use of Precedent	136
	Integrated Design and Analysis	137
	Future Directions of Research	138
	Acoustic Sculpting	138
	Object-oriented Modeling of Architectural Design	139

APPENDICES

A	ACOUSTICAL DATA SOURCE	141
B	COMPUTER CODE FOR THE DESIGN SYSTEMS	143
REFERENCES		214
BIOGRAPHICAL SKETCH		221

LIST OF FIGURES

Figure

1	The mapping of an object (virtual computer) onto a physical computer.	13
2	Encapsulation of data and operations in an object.	15
3	Information hiding in an object.	16
4	Model of an object showing the object's functionalities based on context.	17
5	Computation as communication in object-oriented computing.	18
6	Polymorphism in message sending.	20
7	Class and instance in object-oriented computing.	22
8	Hierarchy of classes and subclasses in object-oriented computing.	23
9	Top-down hierarchy of procedures as a "tree" structure.	28
10	Hierarchical flow of control in structured procedural computing.	29
11	Examples of structures of increasing complexity.	30
12	A procedure as input-output mapping.	31
13	The object as a state-machine.	33
14	Single thread of control in structured procedural computing.	36
15	Multiple threads of control in object-oriented computing.	37

16	Decision tree showing a decision path.	46
17	State-action graph of a problem space.	48
18	An example of a simple option graph with constraints.	50
19	Energy impulse response graph (adapted from Siebein, 1989).	76
20	Model of the proscenium-type auditorium.	79
21	Determination of the wall splay angle from the seating area.	80
22	Elliptical field implied by reflected sound rays.	82
23	Section through the auditorium showing the different parameters.	84
24	Topology of the proscenium-type auditorium.	86
25	Relationships of key parameters in the auditorium model.	88
26	Class hierarchies of computational objects in the system.	94
27	Relationship of performance, proscenium and stage parameters.	96
28	Relationship of input parameters.	97
29	Relationship of parameters that define the balcony.	101
30	Relationships to compute acoustical parameters.	102
31	Printout of the computer screen showing the result produced by the design system for rectangular proscenium-type auditoria using the Boston Symphony Hall parameters.	106
32	Comparison of the results produced by the design system for rectangular proscenium-type auditoria using the Boston Symphony Hall parameters.	107
33	Printout of computer screen showing the result produced by the design system for proscenium-type auditoria using the Kleinhans Hall parameters.	109
34	Comparison of results produced by the design system for proscenium-type auditoria using the Kleinhans Hall parameters.	110

35	Printout of computer screen showing result produced by the design system for proscenium-type auditoria using the Music Hall parameters.	114
36	Comparison of results produced by the design system for proscenium-type auditoria using the Music Hall parameters.	115
37	Printout of computer screen showing result produced by the design system for proscenium-type auditoria using the Theatre Maisonneuve parameters.	117
38	Comparison of results produced by the design system for proscenium-type auditoria using the Theatre Maisonneuve parameters.	118
39	Architectural design as the synthesizing interaction of physical and conceptual entities modeled as computational objects.	122
40	An example of a simple column object.	124
41	An example of a simple grid object.	125
42	Graph representation of a circulatory system.	126
43	Dual representation of a graph.	127
44	A visual program.	128
45	A visual program in three dimensions.	129
46	Printout of the screen of a Macintosh computer showing the desktop metaphor.	130
47	Models of a library using channel-agency nets (after Reisig, 1992).	131

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

THE APPLICATION OF OBJECT-ORIENTED COMPUTING
IN THE DEVELOPMENT OF DESIGN SYSTEMS
FOR AUDITORIA

By

Ganapathy Mahalingam

May, 1995

Chairperson: John F. Alexander
Major Department: College of Architecture

This dissertation has a two-part theoretical basis. The first part is that architectural entities like spatial enclosures can be modeled as computational objects in object-oriented design systems. The second part is that spatial forms of auditoria can be generated from acoustical, functional and programmatic parameters. The method used to establish the theoretical basis is the application of the concepts of object-oriented computing in the development of design systems for auditoria. As a practical demonstration of the theoretical basis, two object-oriented design systems for the preliminary spatial design of fan-shaped and rectangular proscenium-type auditoria were developed. In the two systems, the concept of acoustic sculpting is used to convert acoustical, functional and programmatic parameters into architectural parameters used in the spatial design of the auditoria. Statistical, analytical and mathematical methods are used to generate the spatial forms of the auditoria based on the

various parameters. The auditoria are modeled as parametric computational objects. The implementation of the systems is described in detail. The systems are true design systems because they involve the creation of spatial information from nonspatial information. The application of acoustic sculpting in the implemented systems is tested with case studies. The results are presented and discussed. These systems serve as indicators of the potential of object-oriented design systems in architecture. The dissertation concludes with a projection of how the object-oriented computing paradigm can benefit the creation of design systems in architecture. Future directions for research and development are outlined.

CHAPTER 1 INTRODUCTION

Field of Inquiry

The field of inquiry for this dissertation is situated in the common ground between the fields of computer science and architectural design. This statement assumes that there is a common ground between the fields of computer science and architectural design. Upon a cursory examination of the subject matter of these two fields, it seems that they are not related. Kalay (1987a) distinguishes between the processes of design and computation thus:

Design is an ill-understood process that relies on creativity and intuition, as well as the judicious application of scientific principles, technical information, and experience, for the purpose of developing an artifact or an environment that will behave in a prescribed manner. Computable processes, on the other hand, are, by definition, well understood and subject to precise analysis. They are amenable to mathematical modeling, and can be simulated by artificial computing techniques. (p. xi)

By his contrasting definitions of design and computable processes, Kalay raises the issue of the computability of design. Kalay asks the question, can the process of design be described precisely enough to allow its computation? Kalay's question implies that a precise definition of the design process is necessary before it can be made computable. Different computational paradigms have been used to interact with the computer and process information.¹ Each

¹Data are processed on the computer to create information. Reference is made to information being processed on the computer rather than data because, to the user, the computer is processing information.

computational paradigm has its own characteristic way in which an information processing task is modeled and executed on the computer. Earlier computational paradigms are procedurally biased. In earlier computational paradigms like the structured procedural computing paradigm, it is necessary to articulate an information processing task as a precise hierarchy of procedures before it can be executed on the computer. With emerging computational paradigms like the object-oriented computing paradigm, it may no longer be necessary to procedurally structure an information processing task.

The intent of this dissertation is to explore the application of the object-oriented computing paradigm in the development of computer-based design systems for architectural design. The dissertation tries to establish that architectural design, a subset of design, can be made computable by the application of the object-oriented computing paradigm. The approach used does not require architectural design to be defined as a precise hierarchy of procedures. The precise definition of architectural design has been a problematic endeavor, as is described later in this chapter.

Computable Processes and Architectural Design

To define the common ground between computable processes and architectural design, it is necessary to understand the nature of these two processes. The compatibility of the two processes will determine the computability of architectural design. It is necessary to map the architectural design process onto a computable process or a set of computable processes to achieve the computation of architectural design. The effectiveness of the

mapping will determine the extent to which computer-based architectural design systems can be developed.

Computers and computable processes

The computer is, at a fundamental level, an organized machine that controls the flow of electronic charge. What makes the computer extremely useful is the fact that the presence or absence of electronic charge can represent a unit of information. The control of the flow of electronic charge becomes the processing of units of information.² The presence and absence of electronic charge are commonly characterized in computer science as the binary states of "1" and "0," respectively. Computation occurs at a fundamental level when these binary states are transformed into each other through binary switching elements. The transformation of these binary states involves the flow of electronic charge. Computation, at a higher level, is the control of this flow to process information. The electronic flux is information flux. In a computer, according to Evans (1966), information is represented by binary symbols, stored in sets of binary memory elements and processed by binary switching elements. Binary switching elements are used to construct higher logic elements such as the AND gate and the OR gate. Logic elements are used to perform logical operations in computation. Combinations of logic elements are used to perform complex computational tasks.

Even with a limited repertoire for manipulating information represented electronically, many diverse tasks can be performed on the computer. This is because most kinds of information can be represented as systems of binary states. For example, images can be

²Units of information are often referred to as data.

represented as bit-mapped graphics. Besides, the power of the computer to process various kinds of information is augmented by the range of electrically driven devices that have been developed as computer peripherals. All information processing on the computer has to be done with the basic means of manipulating electronic charge and their permutations and combinations. Therefore, in order to process information on the computer, the information processing task must be represented in a mode that is linked to electronic signals and their characteristic processing methods. The information processing task has to be represented in a systemic manner and amenable to analysis. The ideal model for this representation is one that utilizes the architecture of the computer itself. Limitations in the representation of information processing make it possible only for certain kinds of tasks to be modeled on the computer.

The question is, is architectural design one of them? If it is, how should architectural design be modeled as a computable process? The object-oriented computing paradigm provides the model of synthesizing interaction of computational objects to attain this goal. The power of the object-oriented computing paradigm lies in the abstraction of information processing as interacting virtual computers that are mapped onto a physical computer. Each component of the information processing task utilizes the full architecture of the host computer in the object-oriented computing paradigm.

The architectural design process

The architectural design process is enigmatic at best. It is a difficult process to define. It ultimately involves the transformation of the natural and built environment by the application of knowledge and technological skills developed through sociocultural processes.

The architectural design process results in the intentional transformation of the natural and built environment. It encompasses the sequence of activities from the initial will or intent to the creation of an architectural design embodied in representations.

There has been a constant debate about the nature of design methods begun during the 1960s and continuing ever since. Design has been characterized by Cross (1977) as the tackling of a unique type of problem quite unlike scientific, mathematical or logical problems. He has stated that design problems do not have a single correct answer, generally do not require the proof of a hypothesis and do not aim principally to satisfy the designer's self-imposed goals and standards. Yet design problems contain aspects of the types of problems that do contain those characteristics. Others have defined the design process as a goal-directed, problem-solving activity (Archer, 1965), the conscious effort to impose meaningful order (Papanek, 1972) and the performance of a complicated act of faith (Jones, 1966). These definitions can be characterized as methodological, managerial and mystical points of view, respectively. Cross comments that these definitions contain some truth about what it means "to design," but each definition does not contain all the truth. Cross concludes that no simple definition can contain the complexity of the nature of design. Archea (1987) has challenged the very notion of design as a problem-solving activity by calling design "puzzle making." The range of opinions regarding the nature of design reflects its enigmatic nature.

To articulate the architectural design process that is a subset of design, a question can be posed--what is it that architects do? Architects are involved in the task of designing the built environment from the scale of a single room to that of a city. When architects design, they make decisions about the form and spatial arrangement of building materials and

products that define physical structures and spatial environments. These decisions are made using both intuitive and rational methods. The physical structures and spatial environments that architects design create a complex synthesis of visual, aural and kinesthetic experiences. The goal of many architects is to create interesting and safe environments to facilitate a wide range of positive human experiences. Architects are also actively involved in the sequence of activities required to realize³ their designs through the building construction process.

Another question can be asked--what do architects create when they design? The simple answer to this question is that architects create representations of physical structures to be built and spatial environments to be created. These representations traditionally include drawings, physical scale models and written specifications. They are a mix of graphical, physical and verbal representations. The development of computer technology in the last three decades has enabled computer-based drawings and models to be included in the architect's range of representations. All these representations define a virtual world in which analogues of physical structures and spatial environments to be realized can be manipulated as desired. Architects dwell in the virtual world of their representations. One of the major tasks of an architect is to coordinate different representations such that they all refer to a self-consistent whole yet to be realized.

From the answers to the preceding two questions it becomes clear that when architects design, they make decisions about the form and spatial arrangement of building materials and products that define physical structures and spatial environments and create various representations to communicate the physical structures and spatial environments. The

³The word realize is used in the sense "to make real."

relatively active part of the architectural design process is the making of architectural design decisions, and the passive part is the making of architectural representations. This is a difficult distinction to make because the making of architectural design decisions cannot be easily separated from the making of architectural representations. The making of architectural representations commonly includes the processes of drawing and making models. The process of drawing involves visual thinking, and the process of making models involves physical thought. Visual thinking has been discussed extensively by Arnheim (1969) and McKim (1980). Physical thought is the focus of the deliberations of the Committee on Physical Thought at Iowa State University's College of Design. When an architect is designing, it is very difficult to separate the moment of making an architectural design decision from the representational act that reflects the decision. It is not as difficult to make this separation when the architectural design process occurs on the computer. First-order computer-based design systems in architecture aid the process of making architectural design decisions. Systems that aid the making of representations to communicate architectural designs are second-order computer-based design systems. This aspect is elaborated upon later in this chapter.

The Common Ground

The making of architectural design decisions and the making of architectural representations result in the creation of spatial information. Spatial information is information that defines physical structures and spatial environments. This information can be graphical, physical or verbal. Spatial information has been traditionally conveyed in the form of

drawings. These drawings have been two-dimensional depictions of three-dimensional building components and space through systems of projections and notational conventions. Scale models that are themselves three-dimensional physical structures and define spatial environments have also been traditional vehicles for conveying spatial information. Both drawings and scale models are analogues of the physical structures to be built and spatial environments to be realized. The use of the computer to generate and manipulate spatial information is just the use of another device to create analogue representations of architectural designs. Architects transform nonspatial and preexistent spatial information into new spatial information through the architectural design process. This transformation is at the core of the architectural design process. Since computers can process information represented electronically, the common ground between computer science and architectural design lies in the area of creating and processing spatial information.

Mitchell (1990) has provocatively defined design as the computation of shape information needed to guide the fabrication or construction of an artifact. Mitchell elaborates his definition of shape information to include artifact topology, dimensions, angles, and tolerances on dimensions and angles. This definition is narrow and reductionistic. The definition can be expanded to reflect the architect's preoccupation with things other than shapes. Another definition is that design is the computation of spatial information needed to guide the fabrication or construction of an artifact. In the creating and processing of spatial information, computer science and architectural design come together. Computer-based design systems in architecture by definition bridge the fields of computer science and

architectural design. The research and development of first-order computer-based design systems in architecture using object-oriented computing is presented in this dissertation.

Organization of the Dissertation

The rest of this chapter of the dissertation presents distinct ideas from different subject areas. This chapter constitutes what is normally characterized as the review of existing research. This is followed by a chapter that presents a synthesis of the ideas presented in the first chapter. This chapter reflects the creative portion of the dissertation and composes the methodology section of the dissertation. This is followed by a chapter on the results of synthesizing the ideas and methodology in Chapters 1 and 2. The dissertation concludes with a chapter on the benefits of these ideas and future directions of research.

Chapter 1 contains a brief discussion of the origin and development of object-oriented computing. The key concepts of object-oriented computing are discussed with examples. The switch to object-oriented computing is discussed as a paradigm shift. The transition to object-oriented computing is traced. Existing computational models of the architectural design process are summarized. The notion of a first-order, computer-based design system in architecture is explained. Existing computer-based design systems related to the object-oriented computing paradigm are discussed briefly.

In Chapter 2, the concept of acoustic sculpting is introduced. Acoustic sculpting bases the spatial design of auditoria on acoustical parameters. This concept is used to develop a model of the auditorium as a parametric computational object in an object-oriented computer-based design system. Acoustic sculpting makes it possible for acoustics to be a form giver for

the design of auditoria. The development of two object-oriented computer-based design systems for the preliminary spatial design of proscenium-type auditoria is described. These systems reveal the potential of acoustic sculpting and object-oriented computer-based design systems in architecture.

Chapter 3 contains details of the implementation and results produced by the two object-oriented computer-based design systems. Chapter 4 outlines future directions of research in acoustic sculpting and the object-oriented modeling of the architectural design process. A discussion of the advantages of object-oriented computer-based design systems in architecture is also presented.

Origins of Object-oriented Computing

Even as the structured procedural computing paradigm was becoming popular, work being done at the Xerox Palo Alto Research Center (PARC) based on Alan Kay and Adele Goldberg's vision of the Dynabook (Kay & Goldberg, 1977) was defining emerging computer technology. Research at the PARC laid the foundation for expanding the use of computers by defining virtual computers and graphic interfaces to interact with them. The work included the basic concepts of multitasking, windows, scroll bars, menus, icons and bit-mapped graphics. Implementations of these concepts were used to expand the graphic interface to the computer. These implementations spawned the research and development of graphic user interfaces, which have become an important concern of software developers in recent years. The idea of using pointing devices like a mouse or pen to select icons on the screen and to

perform operations on the computational objects represented by those icons⁴ was also a result of the Dynabook effort. These concepts have since become very popular and have been absorbed into the mainstream of computer technology. The main contribution of the Dynabook effort, however, was the development of Smalltalk--the archetypal object-oriented programming environment, which was formally launched in August, 1981 (Byte, 1981). Smalltalk was based initially on the central ideas of Simula, a programming language for simulation developed by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo (Kay, 1977). Smalltalk began as a programming environment that targeted children down to the age of six as users. It graduated into an exploratory programming environment for people of all ages and eventually became a serious programming environment for software development professionals. The Smalltalk programming environment embodies all the concepts of object-oriented computing and is uniformly object-oriented itself. This is the reason for using Smalltalk in this dissertation to explore the application of the object-oriented computing paradigm in the development of computer-based design systems in architecture. For computer enthusiasts, the history of the development of Smalltalk can be read elsewhere (Krasner, 1983).

Key Concepts of Object-oriented Computing

Object-oriented computing is a relatively new paradigm being used in computation that has the potential of rapidly replacing the structured procedural computing paradigm that was the norm in the 1970s. The object-oriented computing paradigm took root in the 1980s

⁴This is the desktop metaphor of the AppleTM/MacintoshTM operating system interface.

and has been hailed by many as the significant computing paradigm of the 1990s. A characteristic set of concepts defines the paradigm. These concepts are discussed in outline form by Smith (1991). There are also numerous text books on object-oriented computing that explain these concepts with different nuances. A summary of the concepts is provided in the rest of this section.⁵

The Object as a Computer Abstraction

The goal of the developers of object-oriented computing was to provide maximum natural interaction with the computer. To achieve this, they developed a computer abstraction called an object. An object is a composite entity of data and operations that can be performed on that data. Before this, the main computer abstractions being used were data structures and procedures. It was felt by the developers of object-oriented computing that people involved in computation would interact more naturally with objects than with data structures and procedures. The object is at a higher level of abstraction than data structures or procedures. This abstraction allows the analysis and creation of systems at a more general level. It is more natural to decompose systems into physical or conceptual objects and their relationships than it is to decompose them into data and procedures. Data structures and procedures are considered to be at a finer level of "granularity" than objects. In what can be considered a hierarchical system, the level of abstraction progresses from data structures and procedures to objects.

⁵The concepts and terminology of object-oriented computing that are discussed in this chapter refer to the Smalltalk programming environment.

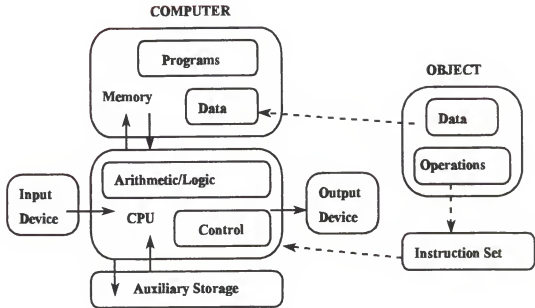


Figure 1. The mapping of an object (virtual computer) onto a physical computer.

The object as a computer abstraction can be mapped onto a physical computer (see Figure 1). In essence, it behaves as a virtual computer that has the full power of the physical computer onto which it is mapped. Each object can be thought of as a virtual computer with its own private memory (its data) and instruction set (its operations). The reference to objects as virtual computers was made by Kay (1977). He envisaged a host computer being broken down into thousands of virtual computers, each having the capabilities of the whole and exhibiting certain behavior when sent a message⁶ that is a part of its instruction set. He called

⁶A message in object-oriented computing is the quasi-equivalent of a function or procedure call in structured procedural computing.

these virtual computers "activities." According to him, object-oriented systems should be nothing but dynamically communicating "activities."

An object in an object-oriented system has also been likened to a software integrated circuit (Ledbetter & Cox, 1985). By extending the concept that objects are software integrated circuits, it is possible to create a set of hardware integrated circuits laid out on a circuit board that represents a software application. A software system for architectural design could conceivably be converted into a circuit board that is plugged into a computer. The object as a computer abstraction enables a modular approach to computation similar to the one used in the design of integrated circuits. A modular approach to computation is not exclusive to object-oriented computing. It has been a feature of programming languages such as Ada and Modula-2, where packages and modules have been used akin to objects. Packages and modules support the concepts of information hiding and data abstraction that are a part of object-oriented computing. However, Ada and Modula-2 are not considered truly object-oriented because they do not support the concepts of inheritance and dynamic binding that are an integral part of object-oriented computing.

Encapsulation

In object-oriented computing, physical and conceptual entities in the world are modeled as an encapsulation of data and operations that can be performed on that data. The data and operations are defined together. Any operation that is not part of this joint definition cannot directly access the data. The concept of encapsulation is also based on the notion of

abstraction. A collection of data and operations performed on the data are closely related, so they are treated as a single entity rather than separately for the purpose of abstraction.

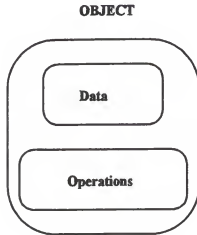


Figure 2. Encapsulation of data and operations in an object.

The bundling of data and operations that can be performed on that data into a "capsule" or computational object is called encapsulation (see Figure 2). This concept is based on the block construct in the structured procedural computing paradigm. Encapsulation enables the concept of information hiding where the data of an object are protected and are accessible only through an interface. Encapsulation enables the abstraction of state in simulation systems developed using computational objects. Encapsulation also enables the concept of polymorphism. These aspects are discussed later.

Information Hiding

The data of an object are private and cannot be accessed directly (see Figure 3). This is the concept of information hiding. The data of an object can only be accessed by the operations of the object. These operations are invoked by sending the object messages. The only way in which you interact with an object is by sending it messages.

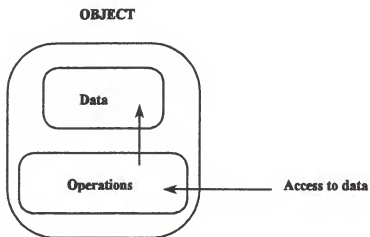


Figure 3. Information hiding in an object.

This interaction is controlled by an interface. The interface is made up of messages that an object understands. Related messages are grouped into protocols. Protocols are used to identify the different functional aspects of the object. Protocols are also used to organize the object development process. When an object receives a message, it invokes the

appropriate method⁷ associated with that message. The interface controls the aspects of the object with which you can interact.

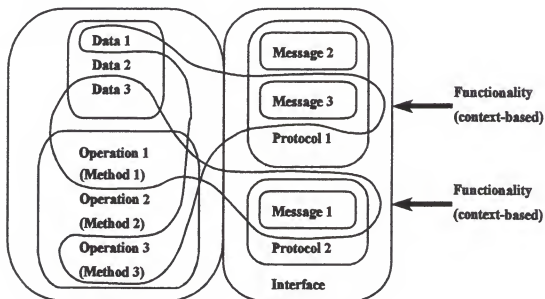


Figure 4. Model of an object showing the object's functionalities based on context.

The interface is another device for abstraction. It can provide several selective modes of interaction with the object. This is an important concept. Selective interfaces to the object can couple different aspects of the object's data with different operations to provide different functionalities for the object. The different functionalities are a result of different mapping operations. An object can behave differently in different modes (see Figure 4). This property begins to move object-oriented computing to the next plateau envisaged by Kay, the creation

⁷A method is the name given to an operation that is part of an object. Each method is linked to a particular message.

of observer languages, where computational objects behave differently based on different viewpoints (Kay, 1977).

Computation as Communication

Computation in an object-oriented system is achieved by objects communicating with each other by sending messages that simulate actual interactions between the objects (see Figure 5). Parallelism is inherent in such a process, as it is in all complex communication systems. Many objects in an object-oriented system can be actively communicating with each other simultaneously. This is because each object is a virtual computer that is mapped onto a host physical computer.

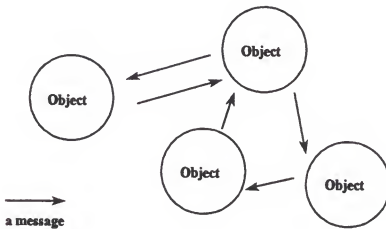


Figure 5. Computation as communication in object-oriented computing.

An object-oriented system has also been likened to a sociological system of communicating human beings (Goldberg & Robson, 1989). By mimicking human communication in the computation process, object-oriented systems make user interaction with the system more natural. In the desktop metaphor of the Apple™/Macintosh™ operating system interface, you can point and click on an icon that represents a file and drag it onto an icon that represents a trash bin to discard the file. Such a natural graphic interaction can be easily modeled in an object-oriented system of communicating objects. The concept of viewing control structures in computation as message sending is reflected in the work of Hewett reported by Smith (1991).

Polymorphism

Through encapsulation, the operations of an object belong exclusively to the object. They do not have an existence outside the object. Therefore, different objects can have operations linked to the same message name. This is the concept of polymorphism. The separation of message and method enables polymorphism. Polymorphism does not cause confusion because the operations are part of an object's definition and can be invoked only through the object's interface. According to Smith (1991), polymorphism eliminates the need for conditional statements like if, switch or case statements used in conventional languages belonging to the structured procedural computing paradigm. Smith (1991) suggests that polymorphism combines with the concepts of class hierarchy and single type in object-oriented computing to provide a powerful tool for programming.

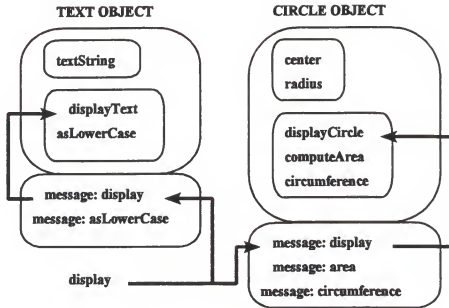


Figure 6. Polymorphism in message sending.

Polymorphism enables easy communication with different objects. The same message can be sent to different objects and each of them will invoke the appropriate method in their definition for that message (see Figure 6). Polymorphism also enables the easy addition of new objects to a system if they respond to the same messages as existing objects.

Dynamic Functionality

An object is dynamic and useful, unlike a data structure, which is static. However, you can only do a few things with an object. You can either query the state of its data or change the data with a message. You can change the state of the data with an externally supplied value, which is usually an argument for a message, or you can ask the object to compute the

change. The object can then change the state of its data with its own operations, or it can request the help of other objects to do it. An object is a dynamic entity because it can represent state and can link to other objects to perform tasks when necessary. An object can represent state because it has a private, persistent memory. The representation of states enables the simulation of objects that change with time, and the capacity to link to other objects increases functionality.

Classes and Inheritance

Objects in an object-oriented system belong to classes for specification or definition. A class is a conceptual tool to model a type of object. A class is a computer definition of a physical or conceptual entity in an object-oriented system. Each object in an object-oriented system is an instance of a class, just as an auditorium is an instance of the class Auditorium (see Figure 7). The system of using classes to define objects is based on the concept of a hierarchy of definitions (Smith, 1991). Classes themselves are objects. They can hold data in class variables and have operations defined as class methods. Class methods are usually used to create an instance of the class. Class variables are used to store global data that can be accessed by all the instances of the class. Abstract classes can also be defined that have no instances. These abstract classes define protocols that subclasses reimplement at their own level or use them directly if they do not override them. Though it may be possible to create instances of abstract classes, the practice is usually discouraged.

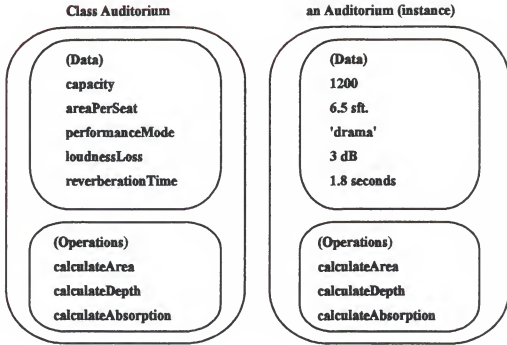


Figure 7. Class and instance in object-oriented computing.

A class comprises data and operations that define the type of object it represents. For example, the class Building would have building components, dimensions, spatial form, etc., as data, and "derive bill of materials," "compute cost," "compute heating load," "compute cooling load," etc., as operations. Class data and operations are general to the class. Every instance of a class has all the data and operations of its class. Subclasses may be hierarchically derived from any class through the mechanism of inheritance. A subclass inherits the data and operations of its parent class, also called a superclass. It can, however, reimplement the data and operations at its level to create a specialized version of its parent class.

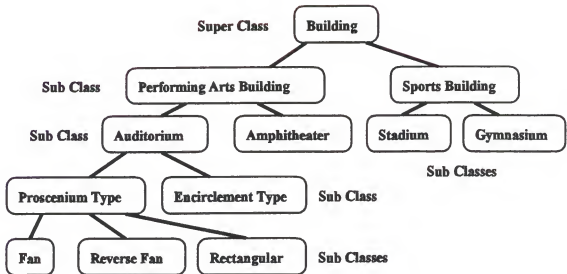


Figure 8. Hierarchy of classes and subclasses in object-oriented computing.

For example, Auditorium and Gymnasium subclasses can be derived from the class Building (see Figure 8). This hierarchical structure allows generalization and specialization in the specification or definition of objects. Some object-oriented languages allow subclasses to inherit from more than one parent class. This is called multiple inheritance. The class structure in object-oriented systems allows the reuse of software components and facilitates programming by extension in software development. To create a new class that is only slightly different from an existing class, one can create a subclass of that class and make the necessary modifications. This facilitates programming by differences in software development. Computational objects representing particular physical or conceptual entities can be reused

or incrementally modified through the mechanism of inheritance. The classification of objects based on similarities and differences is a powerful organizational tool.

Composite Objects

A composite object can be made up of many physical and conceptual objects forming an ensemble. An ensemble can be the model of a complex system. Alternatively, frameworks can also be implemented for the synthesizing of certain types of complex systems. Objects that are unlike each other can be grouped into ensembles that are themselves classes. The behavior of ensembles can be abstracted and modeled. Classes used frequently together for certain kinds of applications can be grouped together in frameworks that can be reused. The design of frameworks involves the design of the interaction between the classes that make up each framework. Ensembles and frameworks are discussed elaborately by Wirfs-Brock and Johnson (1990).

The Paradigm Shift

The increasing popularity of object-oriented computing in the field of computation indicates a paradigm shift as characterized by Kuhn (1962). In the preface to his book on the structure of scientific revolutions, Kuhn (ibid.) defines paradigms as universally recognized scientific achievements that for a time provide model problems and solutions to a community of practitioners. Kuhn further defines paradigms to include laws, theories, applications and instrumentation that together provide models from which spring coherent traditions of research (ibid., p.10). Kuhn states that the development of science is not an incremental

process of accumulation of individual discoveries and inventions, but occurs through paradigm shifts. These shifts can be constructive or destructive. In principle, a new theory might emerge without reflecting destructively on any part of past scientific practice (ibid., p.95). The new theory might be simply a higher level theory than those known before, one that links together a whole group of lower level theories without substantially changing any (ibid., p.95). This is an example of a constructive shift. The paradigm shift from structured procedural computing to object-oriented computing is a constructive shift. In this shift, a higher level of theory subsumes lower level theories. Destructive shifts can happen by discarding some previously held standard beliefs or procedures and, simultaneously, by replacing the components of the previous paradigms with others (ibid., p.66). Though Kuhn was referring specifically to scientific achievements in his work, his notion of a paradigm has come to refer to the core cluster of concepts of any field. In tracing the paradigm shift from the structured procedural computing paradigm to the object-oriented computing paradigm, these core cluster of concepts are discussed.

Building Blocks

The first distinction between the two paradigms is based on their fundamental software components or building blocks. The structured procedural computing paradigm (hereinafter called the procedural paradigm) is so called because the building blocks in structured procedural computing are procedures. In the object-oriented computing paradigm (hereinafter called the object-oriented paradigm), the building blocks are objects. Both objects and procedures are computer abstractions. Data structures are also computer abstractions.

An object is an abstraction at a higher level than data structures or procedures. This is because objects subsume data structures and procedures.

The different levels of abstraction of the building blocks of the two paradigms give the paradigms specific characteristics. In the procedural paradigm, computational tasks are performed in a process-oriented way. Importance is given to a sequence of procedures that are required to perform a computational task. The object-oriented paradigm is problem-oriented, and computational tasks are performed by the interaction of objects that are computer analogues of their real-world counterparts. Importance is given to the objects that are part of the task domain and their characteristics. The objects from the task domain can be physical objects or conceptual objects. The object-oriented approach is a much more natural way of addressing computational tasks because people generally perceive the world around them as comprising objects and their relationships.

Problem Decomposition

The two paradigms can be differentiated by the way in which a computational task is decomposed for execution in each of them. In the procedural paradigm, a computational task is decomposed into subtasks. A sequence of procedures is then developed to perform the subtasks. Each procedure is reduced to subprocedures that have a manageable level of complexity until a hierarchy of procedures has been developed that can perform the computational task. This is called functional decomposition or procedural decomposition. In the object-oriented paradigm, a computational task is decomposed into objects of the task domain and their interaction is structured. This is called object decomposition. Object

decomposition is directly related to human cognition, which perceives its environment in terms of categories (Arnheim, 1969). Object decomposition also enables the abstraction of state in the computational process. This aspect is discussed later.

Top-down Approach versus Unlimited Formalization

The structure of a complex computational task is a hierarchical tree in the procedural paradigm (see Figure 9). This has also been called a top-down approach. At the top of the tree is a procedure that defines the main process in the computational task. This procedure calls other subprocedures to perform subtasks. The subprocedures can call other subprocedures under them. It is a rule that any procedure can only call procedures below its position in the hierarchy. However, a procedure can call itself for a recursive operation. Data are passed down this procedure hierarchy. If the volume of data is high, it becomes very cumbersome to pass it down. If a procedure affects many different data sets, then all these data sets must be passed to the procedure. The solution to avoid passing large data sets to the procedures each time they are called is to make the data global. This lends the data to corruption by the various procedures. If, after a certain procedure has transformed some data, another procedure alters the data in a detrimental way, the end result may be adversely affected. The top-down hierarchical structuring of the procedural paradigm imposes a rigid formalization on any computational task. Circular procedural paths are eliminated in this type of structuring. This limits the modeling of architectural design with this paradigm because there are many circular sequences of decision making in architectural design.

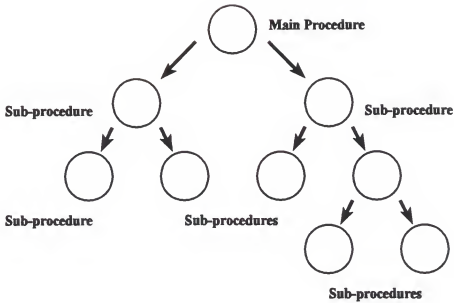


Figure 9. Top-down hierarchy of procedures as a "tree" structure.

Voluminous data are usually made global so that they can be accessed by any procedure at any time. The thread of control passes down a branch of the tree and back up again to flow down another branch. This top-down structuring of procedures with the hierarchical flow of control (see Figure 10) makes it difficult to map data flow diagrams onto the structure. Complex data flows are mapped onto this structure only by using global data that can be accessed by any procedure at any time. With the ease of mapping data flows comes the risk of corruption of the global data. A constant check of the global data must be made to prevent data corruption. This is an additional burden in the procedural paradigm. In large systems, when there are too many procedures, this can become a serious problem.

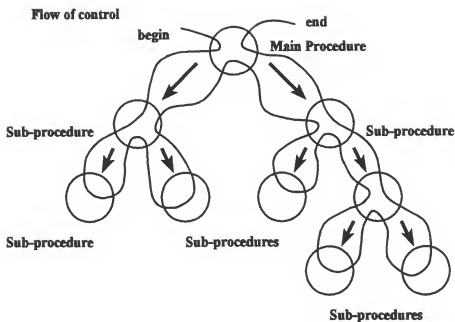


Figure 10. Hierarchical flow of control in structured procedural computing.

In the object-oriented paradigm, there is no top-down hierarchical structure. All objects are on an equal footing with other objects. The structure of a complex computational task can be anything. It can be a tree, a semi-lattice or a network. Examples of these structures are shown in Figure 11. This gives the paradigm the capacity for unlimited formalization (de Champeaux and Olthoff, 1989). The capacity of unlimited formalization means that any formal organizational structure can be adopted in the object-oriented paradigm. The paradigm does not force a particular structure onto a computational task. The most common structure of a computational task in the object-oriented paradigm is a network of objects. Because of unlimited formalization, a structured hierarchy of procedures can also be implemented in the object-oriented paradigm.

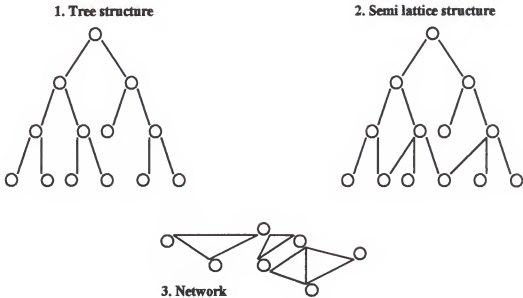


Figure 11. Examples of structures of increasing complexity.

Dijkstra (1972) has said that the art of programming is the art of organizing complexity. The object-oriented paradigm with its unlimited formalization can organize all levels of complexity. The network is a strong candidate for structuring complexity in any system. This is evident by the reasonable success achieved by researchers who have modeled the complex neural architecture of the brain as a network.

Encapsulation versus Data Independence

A procedure is a set of logically related operations that takes input data and transforms them to output data. It is like a black box that does input/output mapping (see Figure 12). The input data are usually passed to the procedure as an argument or parameter

list when a procedure call is made. The input data for a procedure can alternatively be a part of globally available data, i.e., data stored as global variables. A procedure is always dependant on external sources for data. A procedure is an algorithmic abstraction that acts on data stored elsewhere or is passed to it. The data are independent of the procedures that act on the data. Because the data are independent of the procedures that act on the data, the state of the data cannot be abstracted easily. This is a drawback when you try to simulate systems that involve the abstraction of state. In the procedural paradigm, special effort must be made to abstract the state of the data through cumbersome procedures.

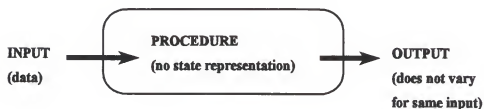


Figure 12. A procedure as input-output mapping.

An object is an encapsulation of data and operations that can be performed on that data. Most of the data that the operations of an object need are stored as a part of the object.

However, an object can also receive data from external sources as message arguments and use them in its operations. For example, an address book can be modeled as an object. The address book object will have an internal memory that is a list of addresses. This is its data. It will also have a set of operations such as "add," "delete" and "look up" that manipulate the data to add an address, delete an address and look up an address. The operations are linked to messages that form a protocol for interacting with the object. When a message is sent to the object, the corresponding operation is invoked. To add an address to an address book object, a message is sent to it to add an address with an address as the argument for the message. The address book object then performs the operation to add the address to its list of addresses. The list of addresses always belongs to the address book object and cannot be directly accessed by any other operation. This protects the list of addresses from being changed by operations belonging to other objects. In contrast, if the address book was implemented in the procedural paradigm, such protection of the data will not be possible unless special efforts are made to restrict the access of the data to qualified procedures. Special efforts will also have to be made to abstract the state of the data relevant to the procedures manipulating them. In an object-oriented system, each object needs a slice of the computer's persistent memory to store its data. Consequently, in large systems, the memory resources needed to have many objects active concurrently can be a problem. Object-oriented programming environments like Smalltalk have "garbage collection" methods to salvage the memory of objects not in use in order to mitigate this problem. These "garbage collection" methods remain constantly active in the Smalltalk programming environment.

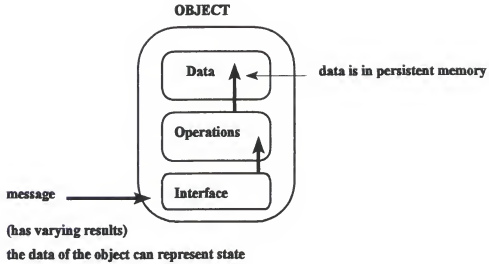


Figure 13. The object as a state-machine.

The encapsulation of data and operations in an object enables the concept of information hiding, polymorphism and the abstraction of state. Because an object encapsulates state, which is represented by its data, and behavior, which is represented by its operations, it has been likened to a "state machine" by Seidewitz and Stark (1987). When a procedure is supplied a certain input like arguments, parameter lists and global data, it always generates the same output. In the case of an object, because it has an internal state, the same input might produce different outputs at different times (see Figure 13). This allows the abstraction of state in the computational process.

Information Hiding

In the procedural paradigm, most widely used data are usually stored globally and can be accessed by any procedure. The data are not protected. This lends it to easy corruption by other procedures. Information hiding prevents data corruption. In the object-oriented paradigm, the data of an object are private and can be accessed only by its operations. No other object can directly access an object's data or directly invoke its operations. This is the concept of information hiding. If an object wants another object to perform an operation or supply data, the object sends the target object (receiver) a message that requests the operation or data. The target object (receiver) then invokes the appropriate operation related to the message or supplies the required data. In this system it is very difficult to corrupt the data.

Static Typing and Dynamic Binding

In structured procedural computing, data and operations on the data are considered separately. This causes a problem. Each procedure must make assumptions about the type of data it is to manipulate. If a procedure is supplied the wrong type of data, an error is generated. Data types include the short integer, the long integer, the floating point, the long floating point, the string and the array. For example, in a process to sort strings, if the procedure is supplied with data representing arrays instead of data representing strings, an error will result. In the procedural paradigm, it is not possible to write a procedure that can sort any type of data. To make sure that a procedure gets the right type of data as input, the concept of data typing has been developed. Type checking ensures that the right type of data

is sent to each procedure. The explicit prescription of a data type for a procedure is called static typing. If explicit types cannot be prescribed, variant records can be used to specify a range of allowable types. In a strongly typed language, the data types for all procedures are known at compile time. Efficient procedural computing needs to be strongly typed.

Object-oriented programming languages can be strongly typed (Eiffel) or typeless (Smalltalk). Type checking in the object-oriented paradigm must not only check the data of the object but also the operations that are permissible. In strongly typed object-oriented languages, only those messages are allowed that can be predicted to be resolved at run time. In the object-oriented paradigm, type checking is more complicated because of the concept of polymorphism. In procedural computing, if a single operation is to be performed on various data types, a global procedure is written with case statements that cover the entire range of data types. If a new data type is added to the system, the case statement must be revised in the procedure. An alternative to this is to have the same procedure written afresh for each data type and make sure that the right procedure is called for each data type. In object-oriented computing, the operation that represents a particular behavior is given the same message name in objects that have data of different data types. It is the responsibility of the object to implement the operation linked to this message to suit its data type. Thus, different objects respond differently to the same message. For example, the message print can be sent to an object that represents a line or an object that represents a character. Those objects would then use their own methods to complete the print operation. This concept, where the same message is sent to different objects to produce different results, is the previously discussed polymorphism. This is made possible by dynamic binding. Dynamic binding means

that the operation associated with a particular message is determined based on the object receiving the message at run time. The drawback of dynamic binding is that errors can only be detected at run time.

Serial Computation versus Parallel Computation

There is a fundamental difference in the way in which these two paradigms treat the computer. The procedural paradigm treats the computer as a serial processor and arranges the program to have a single linear thread of control that passes from procedure to procedure down the hierarchy of procedures and back up again (see Figure 14). Parallelism can be mimicked in the procedural paradigm using co-routines and interleaved procedures. Such parallelism still has a sequential thread of control.

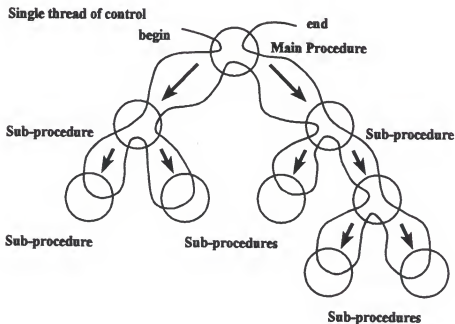


Figure 14. Single thread of control in structured procedural computing.

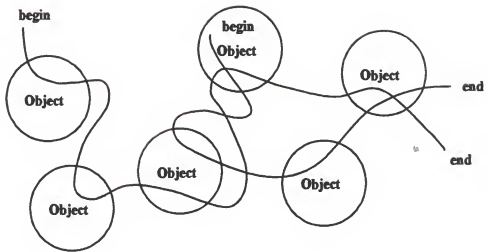


Figure 15. Multiple threads of control in object-oriented computing.

The object-oriented paradigm maps the host computer onto thousands of virtual computers, each with the power and capability of the whole. Each virtual computer or object is constantly ready to act, therefore, the system is inherently parallel. There is no central thread of control in an object-oriented computation. There may be many threads of control operating simultaneously. This is shown in Figure 15. Parallel systems can be implemented using the object-oriented paradigm.

Classes and Inheritance

This concept is unique to object-oriented computing. Two of the main problems in software development are the reuse of software components and the extension of existing

software systems. The class structure in the object-oriented paradigm allows the reuse of software components and supports programming by extension. To create a new class that is only slightly different from an existing class, one can create a subclass of that class and make the necessary modifications. This is the mechanism of inheritance described earlier in this chapter. Inheritance allows programming by incremental differences. Some object-oriented languages allow subclasses to inherit from more than one parent or super class. This is called multiple inheritance. This allows hybrid characteristics to be incorporated in the software components when reused. In the procedural paradigm, procedures can be reused only if they are generic and stored in libraries.

Analysis, Design and Implementation

In the procedural paradigm, the three stages of software development, namely, analysis, design and implementation are disjointed. In the analysis stage, data flows are organized. In the design stage, a hierarchy of procedures is developed. The implementation stage involves the mapping of the data flows onto the hierarchy of procedures using control structures. The changing point of view in the three stages makes the coordination between them very difficult. This affects productivity in the development process and hinders the rapid development of prototype systems.

In the object-oriented paradigm, the focus of interest is the same in all three stages of software development. It is objects and their relationships. Objects and their relationships identified in the analysis stage form the first layer of the design stage and organize the system architecture in the implementation stage. This results in high productivity in the development

process and facilitates the rapid development of prototype systems. This is why the object-oriented paradigm has been hailed as a unifying paradigm (Korson & McGregor, 1990).

The Transition to Object-oriented Computing

The transition to object-oriented computing can be traced as an evolutionary change in the way in which programmers have interacted with the computer to perform computational tasks. In the earlier techniques of programming with high-level languages, instructions to perform a computational task were written sequentially with numerous GOTO statements^{*} to move from one instruction to another, usually in a random manner. A program written in this manner is referred to as spaghetti code because the sequence of instructions to be executed is a tangled web like spaghetti in a bowl. Refinement of this technique resulted in the development of branching and looping constructs. These constructs are used to structure a sequence of instructions into procedures. Procedures are a logically related set of instructions and are treated as independent modules. Continuing the evolutionary trend, branching and looping constructs were applied to procedures to prevent spaghetti modules or a tangled web of procedures. The systematic organization of procedures led to structured procedural computing. The next stage in the evolution resulted in a shift from the use of procedures that act on global data to data packaged with procedures using different constructs. A construct that emerged was the block structure where a block contained a procedure or a set of procedures within which data were protected. The data used only within a block in the form of local variables are not known outside the block. The combination of

^{*}These are programming constructs to move from one instruction to another.

data and operations performed on that data led to data abstraction. According to Wegner (1989) data abstractions are computational constructs whose data are accessible only through the operations associated with them. When the implementation of these operations is hidden to the user, the data abstraction is called an abstract data type. For example, a stack, which is a programming construct, is a data abstraction. When the "push" and "pop" operations performed on the stack do not reveal the implementation of the stack as a list or an array, the stack is called an abstract data type. An abstract data type is a programming construct where a type of data and operations that can be performed on that data are defined together. The data type's implementation is hidden, and the data can be accessed only through a set of operations associated with it. The use of abstract data types has resulted in what Wegner calls object-based computing (Wegner, 1989).

Object-oriented computing is the last stage in a transition that has moved from a purely procedural approach to an object-based approach and then to an object-oriented approach. In the procedural approach, the individual software components are the data structure and the procedure. In the object-based approach, the individual software component used is an abstract data type (ADT), and inheritance is not supported. In the object-oriented approach, the object is the individual software component. The object has a tighter coupling of data and functions than a traditional ADT. In the object-oriented approach, inheritance is supported. A detailed comparison of the three approaches has been given by Wegner (1989). The use of ADTs in computer-aided design systems has been advocated by Eastman (1985). Some of Eastman's current ideas on building modeling seem to belong to the object-oriented

paradigm though he takes care to distinguish it as being different from object-oriented computing (Eastman, 1991).

Computable Models of Architectural Design

The architectural design process can be defined as a two-part process. The first part of the process is making decisions⁹ about the form and spatial arrangement of building materials and products that define physical structures and spatial environments. The second part of the process is making various representations that communicate those structures and environments. The process of making architectural design decisions cannot be easily separated from the process of making architectural representations because of visual thinking and physical thought. Visual thinking occurs during the process of making drawings, and physical thought occurs during the process of making physical models. This aspect has been mentioned earlier. However, in a computer-based architectural design process, architectural design decision making can be separated from the making of architectural representations. With the rapid development of computer technology, computable models have been constantly sought to simulate the entire architectural design process but with little success. However, many computable models have been developed for clearly identifiable parts of the architectural design process. These models computationally assist parts of the architectural design process or make them computable. Computable models of parts of the architectural design process represent design activities as information processing tasks on the computer using available

⁹This includes all the processes that help make the decisions like research and analyses.

computer abstractions. This representation has included both the activities of making architectural design decisions and making architectural representations. Computer models for representing architectural objects and environments have been elaborately discussed by Kalay (1989) in his book on the modeling of objects and environments. Some of the key models of architectural design decision making have also been discussed by Kalay (1987a) in his book on the computability of design. These models are summarized in the rest of this section.

Computable Models for Making Architectural Representations

Computable models for making architectural representations were the earliest to be defined. The process of creating architectural representations on the computer is a superset of creating graphics on the computer. It uses all of the representational models available in computer graphics. The process of drawing, which is the most common way to create architectural representations, is modeled as the synthesis of primitive elements such as lines, arcs, splines and shape elements such as circles, ellipses and polygons. Actually, the arcs, splines, circles, ellipses and polygons are made of tiny line segments reducing the computable model of drawing, in effect, to the synthesis of lines. Alphanumeric text is also available in this model to annotate the drawings and to create verbal representations. Lines and alphanumeric text form the basic elements of a computable model of drawing. Translation, rotation and scaling are typical operations available to manipulate these basic elements. In the computable model of drawing, lines and alphanumeric text are combined in Cartesian space. The synthesis is usually an aggregation of the lines and alphanumeric text in the order that they are created. To add complexity to the model, lines and alphanumeric text can be grouped together into

symbols that can then be manipulated as individual entities. Areas bounded by lines, arcs, splines, circles and polygons can be filled with colors or patterns to indicate different materials. Many different aspects of a representation can be overlaid in a computer-based drawing using the concept of layers. No specific structure is maintained in the computer-based drawings other than the structures implied by symbols and layers. The drawing is stored as a database file containing records for the individual elements. The lack of meaningful structure¹⁰ in computer models of drawings has been discussed by Mitchell (1989). Embedded subshapes have been proposed by Tan (1990) to add meaningful structure to computer-based drawings. This allows the open interpretation and semantic manipulation of a computer-based drawing. The computable model of drawing as it is embodied in conventional computer-based drawing or drafting systems does not allow the manipulation of the drawing based on visual thinking. This is because visual thinking involves perceptual subunits in the drawing that are not explicitly stored as a part of the drawing's structure. Embedding subshapes in a computer-based drawing is a strategy to overcome this limitation.

Another aspect of the creation of architectural representations is the modeling of three-dimensional objects. Different computer models have been developed to represent three-dimensional objects. These are discussed in detail by Kalay (1989). Constructive Solid Geometry (CSG) represents solid objects as Boolean combinations (union, intersection and difference) of a limited set of primitive solids like cubes, cylinders, wedges, spheres and tori. A complex solid is stored as a binary tree. The terminal nodes of the binary tree contain

¹⁰The structure of the drawing is the aggregation of the basic elements of the drawing. This does not allow meaningful perceptual subunits of the drawing to be manipulated.

primitive solids or transformed instances of the primitive solids. The nonterminal nodes of the binary tree contain linear transformation operators (rotation and translation) or Boolean operators (union, intersection and difference). In CSG, other advanced operations such as sweeps and extrusions of primitive solids are also used to generate complex three-dimensional objects. However, the CSG model is inefficient when the boundary surface of the object is needed in applications. The Boundary Representation (B-rep) model represents a solid as a set of faces that form the bounding surfaces of the solid. The B-rep model is also called polyhedral representation. This model comprises geometric and topological information. The geometric information supplies the dimensions and spatial location of the elements that make up the bounding surface. The topological information supplies the relationships or connectivity among those elements. The B-rep model uses an edge-based data structure and Euler operators to create the boundary representation of solids. There are many variations of the edge-based data structure like the winged edge, the split edge and the hybrid edge which have been explained by Kalay (1989). The faces of a B-rep model can be shaded in any color or have a texture or pattern mapped onto them because they behave like polygons. This allows the solids in the B-rep model to simulate different materials under different light conditions making it possible to create architectural representations. Another representational model available in computer graphics is ray tracing which is used to create realistic representations of architectural designs (Glassner, 1989). Ray tracing is used to build an image by tracing rays of light from the eye onto the physical objects that make up the image.

Computable Models of Architectural Design Decision Making

Many computable models have been proposed for architectural design decision making. Only some key models are presented in the rest of this section. These models have driven the development of computer-based design systems in architecture. They also represent a progression in the way in which architectural design decision making has been modeled to make it computable.

Problem solving

One of the earliest models to be adopted to make architectural design decision making computable was the problem-solving model. Research done by Newell and Simon (1972) defined this model clearly enough for it to be adopted in many fields of human decision making. Newell and Simon's research (1972) on human problem solving influenced the consideration of design as a problem-solving process to a great extent. Simon himself acknowledged in a later study (1973) that design is an ill-structured problem-solving process. However, the computability of design being dependent on the consideration of design as a problem-solving process has been maintained (Kalay, 1987a). This view is linked to the procedural paradigm. In the past, it may have been necessary to conceive of design as a problem-solving process to make it computable, i.e., to fit the process-oriented procedural paradigm. The state-action graph model (Mitchell, 1977) and the decision tree model (Rowe, 1987) of design as a problem-solving process clearly illustrate this aspect when they are compared to the top-down hierarchical tree of procedures in the procedural paradigm. It is not clear if the characterization of the design process as a problem-solving activity based on

decision trees and state-action graphs was influenced by computational models that were prevalent at that time.

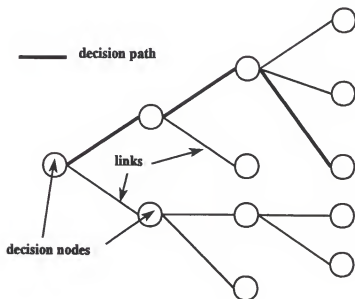


Figure 16. Decision tree showing a decision path.

The problem solving model of architectural design treats architectural design as a general problem. Simon (1973) explains the requirements of a General Problem Solver in his paper on the structure of ill-structured problems. A General Problem Solver (GPS) has the following five requirements:

- 1) A description of the solution state, or a test to determine if that state has been reached
- 2) A set of terms for describing and characterizing the initial state, goal state and intermediate states

- 3) A set of operators to change one state into another, together with the conditions for the applicability of these operators
- 4) A set of differences, and tests to detect the presence of these differences between pairs of states
- 5) A table of connections associating with each difference one or more operators that are relevant to reducing or removing that difference

These requirements can be resolved into three categories according to Rowe (1987). They are knowledge states, generative processes and test procedures. These requirements together constitute a domain called the problem space. The structure of a problem space is represented as a decision tree. The nodes of the tree are decision points, and the branches or edges are courses of action. By traversing the decision tree of a problem space, a solution can be found to the problem. The path of the traversal defines a particular problem solving protocol (see Figure 16). The state-action graph can be mapped onto a decision tree (see Figure 17). The nodes of the decision tree are occupied by knowledge states. The branches reflect the operations or actions that can be performed on those states. Testing occurs at each node and may be linked to the state of the previous node. If architectural design is to be performed using a GPS, there must be mechanisms that represent

- a) the state of an architectural design,
- b) operators that can change that state and their rules of application,
- c) tests to detect the difference between the states of the architectural design,
- d) operators associated with the removal of differences in those states, and
- e) tests to determine if a solution state has been reached.

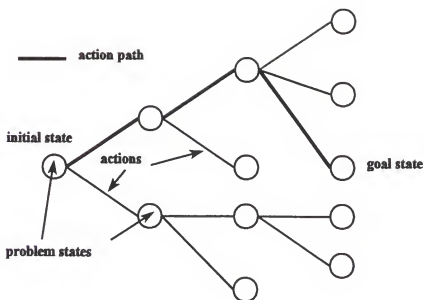


Figure 17. State-action graph of a problem space.

Computable models for making architectural representations provide the mechanism for representing the different states of an architectural design. Operators available in those models can be used as operators in the problem solver if they maintain the semantic integrity of the states they manipulate. Tests on those states can be performed by evaluation mechanisms. Different evaluation mechanisms are presented in Kalay's book (Kalay, 1992) on the evaluation of the performance of architectural designs. There are some fundamental shortcomings in the problem-solving model of architectural design decision making. The classic definition of a problem has been attributed to Thorndike (1931). He stated that a problem exists if something is desired but the actions necessary to obtain it are not immediately obvious. Problem solving is goal-directed activity in the sense that the goal is the

object of desire. According to Mitchell (1977), in order to state a problem, some kind of description of the goal must be provided. In the problem solving model, alternate solutions are generated and tested till a "satisfying" solution is found. The problem-solving approach is based on the assumption that the characteristics of a solution can be formulated prior to engaging in the process of seeking that solution. Decision making in this model becomes a goal-directed activity based on means-end analysis. The drawback of this model is the fact that, in architectural design, the characteristics of a solution are seldom formulated prior to seeking the solution. The characteristics are modified and changed during the process of design.

Constraint-based decision making

Constraint-based decision making evolved to rectify some of the shortcomings of the problem-solving model. Constraint-based decision making allows the addition of new constraints as the decision making progresses. This allows the modification of the goals or objectives of the decision making activity. Constraint-based decision making was applied to architectural design decision making by Luckman (1984) using what he called an analysis of interconnected decision areas (AIDA). He identified certain decision areas in a design task and enumerated the options in each of the decision areas. Then he linked options that were incompatible with each other to arrive at what he called an option graph (see Figure 18). Option graphs are maps of constraints in decision making. An option graph is resolved if all the constraints are satisfied when a set of options is selected. This model lends itself to be implemented in a visual programming language.

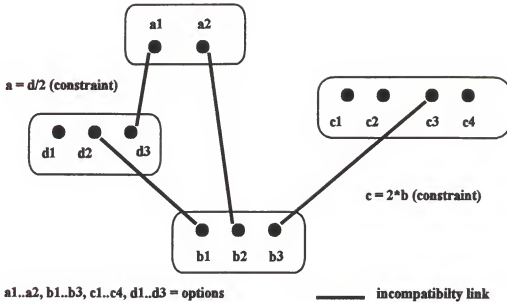


Figure 18. An example of a simple option graph with constraints.

In an option graph, feasible solutions to the design task include an option in each of the decision areas without violating any of the incompatibility links. All the decision areas are on equal footing, so the option graph is not a directed graph with some decisions preceding others. The sequence of decisions is suggested by the pattern of links in the option graph. The option graph may reveal rings or circular paths of decisions. When this happens, the decision making is resolved in the circular paths before branching into other decision areas. When more than one option is available in a decision area, an option is chosen based on other criteria. Incompatibility links and criteria in option graphs are often not deterministic. Probabilistic relationships can be defined in option graphs that require the use of statistical

decision theory in the search for a feasible solution. Option graphs with many links can be resolved only by using a powerful computer because of the combinatorial nature of the problem. Guesgen and Hertzberg (1992) have defined a constraint as a pair consisting of a set of variables belonging to corresponding domains and a decidable relationship between the domains. This is similar to Luckman's incompatibility link. The decision area corresponds to the domain. The variables correspond to options, and the relationship is the incompatibility. Guesgen and Hertzberg also define a constraint network that is similar to Luckman's option graph. According to them, a constraint network is a pair consisting of a set of variables and a set of constraints where the variables of each constraint is a subset of the set of variables. A solution to the constraint network is obtained when every variable is assigned a value and all constraints are satisfied for the value combination. The constraint based decision making model is similar to the problem solving model in that it is goal-directed decision making. The goal in a constraint-based decision making model is the satisfying of multiple constraints. Constraint-based decision making starts with an initial set of variables and constraints that may be incomplete or even contradictory or misleading. As the constraint-based decision making progresses, the model allows the addition of new constraints that narrow the decision making to what is eventually a satisfying solution. This model allows the incorporation of fresh insights in the decision making process and is closer to the way in which architects work. Constraint-based decision making allows the incorporation of circular decision making paths that are not possible in the problem-solving model. The tree structure in the problem-solving model is special kind of graph that does not have circuits, i.e., the nodes and edges of the tree do not form circular links.

Puzzle making

Another model of design decision making is puzzle making. This model should not be confused with puzzle solving which is a kind of problem solving. Puzzle making involves the creation of unique artifacts that are perceptually resolved by the people interacting with them. To enable the resolution of the puzzle, each of its components must be identifiable from prior experience. The perceptual resolution of the puzzle is not immediate because of the unique combination of the components. In architecture, the puzzle making process is characterized as the discovering of unique spatiotemporal environments that can be created by combining architectural elements using rules that are based on precedent. The architectural elements themselves are derived from precedents or created afresh. This model emphasizes the use of precedent and implies that designs are not created from a clean slate. Puzzle making was discussed at length by Archea (1987). In a transition from the problem solving model, puzzle making moves toward an object-oriented approach in its formulation.

Computer-based design systems that serve the first two models involve modules that are used to represent candidate solutions and allow their transformation, and modules that test those solutions to determine if they are satisfactory. Conventional computer-based drawing or drafting systems provide only the representational medium. The analysis and testing of those representations involves the use of additional software. Separate software is also needed to monitor the search process and administer the constraints. Since the representations contain only limited descriptive data, all other required information is stored in a relational database. The coordination between the different modules and the relational database is a cumbersome process. The object-oriented approach with encapsulated state and behavior can

solve this problem. The puzzle-making model of design decision making lends itself directly to the object-oriented approach. The three models presented represent a transition from what can be characterized as a procedural model of architectural design decision making to an object-oriented model of architectural design decision making.

To show that the structure of the problem-solving model was ineffective for complex tasks, Alexander (1965) argued that the naturally growing city is not a tree. He was referring to a hierarchical organizational structure when he used the metaphor of the tree. He meant that the naturally growing city was not hierarchically organized. At the same time, he recognized that "designed" or "artificial" cities were hierarchically organized. Similarly, the natural design process is not a tree. It is not a hierarchically organized sequence of tasks, at least the way it is practiced. Design has been theorized as an artificial process (Simon, 1969). This has been one of the foundations for the development of computer-based design systems. Because the teleological nature of artificial systems is problematic, the design process is not well represented as an artificial system--as a goal-directed problem-solving activity.

In design, there is a constant communication of ideas based on different aspects. Goals are not specified a priori, they are made up along the way. They are modified or changed all the time. Purposes mutate. Physical and conceptual entities are synthesized in this communication process. This dynamic nature of the design process is reflected more accurately by a dense "net" than a hierarchical "tree." Object-oriented computing supports this "net" structuring of the design process. Simon (1969) believed that one of the central structural schemes of complexity was a hierarchy. However, he qualified the definition of "hierarchy" to include systems that resembled the "semi-lattice" structures of Alexander. The

characterization of the design process as a "net" moves toward a more complex and nonformal structuring than "hierarchies" or "semi-lattices."

In the problem-solving model, to use Christopher Alexander's phrase (Alexander, 1965), the design process becomes "trapped in a tree." Constraint-based decision making allows relatively greater freedom in modeling architectural design decision making. Puzzle making allows even greater freedom than constraint-based decision making. The transition from problem solving to constraint-based decision making and then to puzzle making is paralleled by the transition in computing paradigms. Problem solving and constraint-based decision making are best implemented in the procedural paradigm. Puzzle making is best implemented in the object-oriented paradigm. In the object-oriented paradigm, design can be modeled in ways other than puzzle making. It can be modeled as the synthesizing interaction of physical and conceptual entities. This would make the design process less deterministic and more creative.

First-order Computer-based Design Systems in Architecture

Before any discussion of computer-based design systems in architecture begins, there is a need to clarify the meaning of the term CAD. CAD should rightfully stand for Computer-Aided Design. A first-order CAD system should significantly assist in the activity called design. This assistance should be predominantly for the relatively active part of the design process, i.e., the making of design decisions. Systems that predominantly assist the relatively passive part of the design process, i.e., the making of representations, are second-order CAD systems. CAD can also conveniently stand for Computer-Aided Drafting or Computer-Aided

Drawing. Most commercial systems like AutoCAD™, VersaCAD™, DesignCAD™, etc., are predominantly drafting systems. A computer-aided drafting system is one that enables you to create drawings that are representations of designs. The relatively passive act of creating a representation of a design has often been confused with the active process of making design decisions. The confusion is compounded by visual thinking which occurs during the process of drawing, making it difficult to separate the process of making decisions from the process of making representations. For example, a computer-aided drafting system can help you draw the plan for a house but cannot help you determine what the shape of the plan should be. Design decision making is the activity that determines the shape of the plan. The decision making, however, may not occur prior to the making of representations but through it. Computer-based drafting systems are touted as computer-based design systems based on their modeling facility, specifically solid modeling. Solid modeling systems are capable of representing three-dimensional geometric entities and performing transformational and combinatorial operations on them. State-of-the-art solid modeling systems can depict an architectural design in true perspective with almost photographic realism in full color. A modeling system is only a visualization tool that enables the architect to visualize something that has already been designed. It does not help the making of initial design decisions. However, it is an aid to the activity of design development that follows the process of initial design decision making. This is because the visualization offers insight that can modify subsequent design decisions. Conventional commercial CAD systems are excellent for the creation of representations and are good second-order CAD systems. A first-order CAD

system is one that assists in the making of design decisions, or better yet, it is a system that makes design decisions. A similar distinction was made by Yessios (1986).

Architectural design is achieved through a series of design decisions. The goal of the decision making is to enable the construction of physical structures and spatial environments that are within acceptable variations of socially-defined performance standards. Since, generally, there are no specific sequences of decisions to translate a set of requirements or ideas into a design for a built environment, the process of making design decisions is usually not algorithmic. Consequently, it is difficult to develop computer-based systems that automate design.

Existing Systems

A component-based paradigm for building representation based on object-oriented computing has been proposed recently (Harfmann & Chen, 1990). However, that concept is limited because it only considers the modeling of physical objects and not conceptual objects. By modeling only the physical objects, the paradigm will have the same inadequacies as pointed out for solid modeling by Yessios (Yessios, 1987). The call for modeling of conceptual objects is akin to Yessios' call for void modeling. Kalay's WorldView system (Kalay, 1987b) and Eastman's Building Modeling (Eastman, 1991) both belong to the object-oriented paradigm. There are numerous object-oriented design systems developed by researchers for minor applications, but the three mentioned above are relatively comprehensive in their scope.

Methodology of the Dissertation

This dissertation has a two-part theoretical basis. The first part is that the object-oriented paradigm can be applied in the development of computer-based design systems in architecture. The second part is that the spatial form of auditoria can be created based on acoustical parameters. The theoretical basis of the dissertation is established through the development of an object-oriented computer-based design system for the preliminary design of proscenium-type auditoria.

The dissertation includes the following methods:

- a. Methods to correlate acoustical parameters with architectural parameters used in the spatial design of auditoria using the concept of acoustic sculpting
- b. Methods for the design of an object-oriented computer-based design system for the preliminary design of proscenium-type auditoria
- c. Methods to optimize spatial form based on multiple criteria

The methods involved in acoustic sculpting include gathering acoustical data in auditoria of different shapes and sizes; obtaining architectural measurements of those auditoria like widths, heights, seating slopes, volume and surface areas; correlating the acoustical and architectural data statistically; obtaining mathematical relationships using regression techniques and deriving other relationships between acoustical and architectural data based on analytical theory and mathematical modeling. Methods used in the development of the object-oriented design systems for the preliminary spatial design of proscenium-type auditoria include parameterizing the spatial form of the auditoria in terms of the acoustical,

programmatic and functional parameters; developing the algorithms to compute the spatial form of the auditoria and using the object-oriented paradigm to make the spatial form of the auditorium a computational object. Methods involved in the optimizing of multiple criteria in the design of the auditoria initially included spatial optimization techniques using ideas from Boolean operations in solid modeling and optimization by constraints. The methods of spatial optimization using Boolean operations are not implemented in the design system developed. The criterion of focus is acoustics considering the building type being modeled (an auditorium). Programmatic and visual criteria are simply optimized in the design of the auditoria using averages, maxima and minima.

The implemented system explores the common ground between architectural design and computer science. This involves the creation of spatial information from nonspatial information. The spatial correlates or loci of acoustical parameters are used in a macrostatic model rather than a microdynamic model in the design systems developed. The methodology involves statistical correlates, analytical theory and mathematical modeling. The acoustical parameters used are measures derived from sound energy transformed by spatial and material configurations. They are acoustical signatures of the spaces in which they are measured. In the systems, acoustics is a form giver for the auditoria. Other parameters are also form givers for the auditoria. The optimal resolution of the resultant spatial configuration based on the different parameters is at the core of the design system.

CHAPTER 2 METHODS

Acoustic Sculpting

Acoustic sculpting is the creation of spatial forms based on acoustical parameters. It can be likened to sculpting, not with a chisel, but with abstract entities such as acoustical parameters. Acoustical parameters become special abstract tools that shape environments in their own characteristic ways, hence the term acoustic sculpting. In this context, it is interesting to introduce the concept of a locus. In planar geometry, loci are lines traced by points according to certain rules or conditions. A circle is the locus of a point that is always equidistant from a given point. An ellipse is the locus of a point whose sum of distances from two given points is always equal. From these examples, it can be seen that a particular rule or condition can trace a particular locus. The scope of application of the concept of a locus can be dramatically widened by realizing that the word locus in Latin means place. Architectural design involves the creation of representations of places and spaces. A question can be posed, what is the locus of an acoustical parameter? In answering that question, spatial forms based on acoustical parameters can be created. Acoustics can become a form giver for architecture.

Acoustical parameters are often measured to assess the acoustical quality of a space or a scaled architectural model. They are indicators of the acoustical characteristics of the

spaces in which they are measured. However, it is important to realize certain facts about acoustical parameters. Acoustical parameters are location specific. For a given sound source in a room, acoustical parameters vary systematically at different locations in the room. Acoustical parameters also vary when the sound source is varied both in frequency and location. Hence, a set of acoustical parameters at a given location for a specific sound source can be used only to generate the general features of the space around that location. This, to stay within the metaphor of sculpting, will result only in a first cut. Different sets of acoustical parameters from different locations for a particular sound source can further refine the generation of the space encompassing those locations. The spatial forms generated by each set of parameters may have to be optimized using Boolean operators like union, intersection and difference to arrive at the spatial form corresponding to all the parameters. It has been found by researchers that at least 10 to 12 sets of acoustical parameters are required to derive the mean values of acoustical parameters in an auditorium (Bradley and Halliwell, 1989). If spatial forms can be created from acoustical parameters, then a rational basis can be established for the creation of acoustical environments. Acoustical parameters are measures derived from sound energy transformed by the space in which they are recorded. These parameters are in effect the acoustical signatures of the space in which they are measured.

Currently, the creation of acoustical environments is a trial-and-error process that tries to match the acoustical parameters of the space being created, probably in the form of a physical model, with acoustical parameters that have been observed in other well-liked spaces. The manipulations of the spatial form of the acoustical environment to achieve the match are done in an arbitrary fashion with no explicit understanding of the relationships between the

form of the space and the corresponding acoustical parameters. There has been extensive research conducted in the 1960s, 1970s, 1980s and 1990s by Beranek (1962), Hawkes (1971), Cremer (1978), Ando (1985), Bradley (1986a), Barron (1988), Barron & Lee (1988), Bradley & Halliwell (1989) and Bradley (1990) to establish those aspects of the auditory experience that are important in the perception of the acoustical quality of a space and how they relate to measured acoustical parameters in that space. There has not been much research conducted (except by Borish, Gade (1986), Gade (1989) and Chiang (1994)) regarding the relationships between acoustical parameters and the forms of the spaces in which they are generated. The concept of acoustic sculpting attempts to define the latter relationships and uses them to create a system that generates spatial forms of auditoria based on specific acoustical parameters. This generative system is used as a tool for creating preliminary spatial designs of proscenium-type auditoria. The object-oriented paradigm is used to develop the generative design system into a software system which models the spatial form of the auditorium as a parametric computational object.

The Method of Acoustic Sculpting

A systematic procedure has been followed to implement the concept of acoustic sculpting. Acoustical research has been done by a team headed by Gary Siebein at the University of Florida to collect the acoustical data needed to implement the concept of acoustical sculpting. First, acoustical data has been collected in class rooms, lecture halls, multipurpose rooms, churches, auditoriums and concert halls using the methods described in the references in Appendix A. The acoustical data have been transformed into standard

acoustical parameters used in architectural acoustics. Then, specific architectural measurements have been obtained for the spaces in which these acoustical measurements were recorded. These measurements have been manually derived from architectural drawings and scaled illustrations of those spaces. The architectural measurements have then been correlated to the acoustical parameters statistically. Regression equations have been obtained from the statistical relations. The process of generation of the spatial form of the auditorium has been derived using both statistical and analytical methods. All the acoustical parameters for the generative system have been drawn from, but are not limited to, the set presented in the following section.

Acoustical Parameters

The acoustical parameters presented next are the general parameters. Different researchers have used different nuances and derivations of these parameters in their studies. Though the list is extensive, not all of the parameters were used in the design generation stage of acoustic sculpting.

1. Reverberation Time
2. Early Decay Time
3. Room Constant
4. Overall Loudness or Strength of sound source
5. Initial Time Delay Gap
6. Temporal Energy Ratios
 - a. Early/Total Energy Ratio--Deutlichkeit

- b. Early/Late Energy Ratio--Clarity
- c. Late/Early Energy Ratio--Running Liveness
- 7. Center Time
- 8. Inter-Aural Cross Correlation & Lateral Energy Fraction
- 9. Bass Ratio, Bass Level Balance, Treble Ratio, Early Decay Time Ratio and Center Time Ratio
- 10. Useful/Detrimental Ratio, Speech Transmission Index and the Rapid Speech Transmission Index

A detailed description of each of the acoustical parameters is presented next.

Reverberation time (RT). The RT of a room is the time (in seconds) required for the sound level in the room to decay by 60 decibels (dB) after a sound source is abruptly turned off. The 60 dB drop represents a reduction of the sound energy level in the room to 1/1,000,000 of the original sound energy level. RT is frequency dependent and is usually measured for each octave band or one-third octave band. Usually the RT at mid frequency (500 Hz-1000 Hz) is used as the RT of the room. In normal hearing situations, it is not possible to hear a 60 dB decay of a sound source because of successive sounds. Another measure is used to assess the part of the reverberant decay that can be heard called the Early Decay Time. The RT parameter contributes to the subjective perception of "liveness," "resonance," "fullness" and "reverberance." The RT parameter was made significant by Sabine. The quantitative measure for RT according to the Eyring Formula is:

$$RT = - (0.049V / S_T * \ln(1-\alpha))$$

where

V = volume of the room in ft^3

S_T = total surface area of room in ft^2

$\ln$ = natural logarithm

α = mean absorption coefficient of the room

This formula can be used along with a V/S_T table developed by Beranek (1962) to determine α for the auditorium.

Early decay time (EDT). The EDT of a room is the time (in seconds) required for the sound level in a room to decay by 10 dB after a sound source is abruptly turned off. It is usually extrapolated to reflect a 60 dB decay for comparison with the RT. The location to location variation of the EDT is usually greater than the location to location variation of the RT. This parameter is very highly correlated to RT for obvious reasons. This parameter, when the values are small, contributes to the subjective perception of "clarity." (Hook, 1989)

Room constant (R). The R is also known as Room Absorption (RA). It is measured in square feet or square meters of a perfectly absorptive surface whose absorption coefficient is 1.0. The unit of measurement is called a sabin. A sabin is a unit area of perfect absorber. The R or RA is calculated by summing the absorption of all the different surfaces of the room along with the absorption due to occupants and the air in the room for a given frequency band. The absorption of a surface is obtained by multiplying the area of the surface by its absorption coefficient.

Relative loudness (L) or strength of sound source. The overall loudness at a certain location in a room is the ratio in dB of the total sound energy from the sound source received at that location to the sound energy of the direct sound from the same source at a distance

of 10 meters in an anechoic space. This parameter contributes to the subjective perception of "loudness" or "strength." The quantitative measure for L is:

$$L = 10 \log \left(\int_0^{\infty} p^2(t) dt / \int_0^{5\text{ms}} p^2(t) dt \right)_{10\text{m}}$$

where

$p^2(t)$ = squared impulse response

ms = milliseconds

Initial time delay gap (ITDG). The ITDG is the time (in milliseconds) between the arrival of the direct sound at a given location and the arrival of the first reflection at the same location. The time delay gap can also be measured for successive reflections. This parameter contributes to the subjective perception of "intimacy" of a performance according to Beranek (1962). An empirical lower limit of 20 ms for ITDG was established by Beranek (1962). In their recent work, some researchers have found that the ITDG does not correlate to the subjective perception of "intimacy" though the reasons for this are not clear (Hook, 1989).

Early/total energy ratio. This is the ratio in dB of the early sound energy (direct sound plus early reflections) received at a certain location in the room to the total sound energy received at that location. It is measured for different time segments that constitute the "early" time period. The time segments are usually 30 milliseconds (ms), 50 ms, 80 ms and 100 ms. This parameter is also called the Deutlichkeit and was developed by Thiele (1953). This parameter contributes to the subjective perception of "definition," "distinctness" and "clarity." It is important for the intelligibility of speech and music. The quantitative measurement for this parameter is:

$$\text{Early/Total Energy Ratio(Deutlichkeit)} = 10 \log \left(\int_0^t p^2(t) dt / \int_0^{\infty} p^2(t) dt \right) \text{ (Bradley, 1990)}$$

where

$p^2(t)$ = squared impulse response.

t = time segment for the early period

Early/late energy ratio. This is the ratio in dB of the early sound energy (direct sound plus early reflections) received at a certain location in the room to the sound energy arriving at the same location in the later part of the reverberant decay period. This ratio is also measured for different time segments that constitute the "early" time period. The time segments are usually 30 ms, 50 ms, 80 ms and 100 ms. The Early/Late Energy Ratio is also known as Clarity (C), a term given by Reichardt (1981). An inverse of this measure called Running Liveness (RL) was postulated by Schultz (1965). It is a measure of the Late/Early Energy Ratio. The Early/Late Energy Ratio is strongly correlated to EDT but in a negative way. Both these parameters contribute to the subjective perception of "clarity" and to speech and music intelligibility. They are also intended to measure the relative balance between clarity (indicated by the strength of the early reflections) and reverberance (indicated by the integrated reverberant or late energy level). The quantitative measurement for the Early/Late Energy Ratio (Clarity) is:

$$C_t = 10 \log (\int_0^t p^2(t) dt / \int_t^\infty p^2(t) dt) \text{ (Bradley, 1990)}$$

The quantitative measurement for the Late/Early Energy Ratio (Running Liveness) is:

$$RL_t = 10 \log (\int_t^\infty p^2(t) dt / \int_0^t p^2(t) dt) \text{ (Bradley, 1990)}$$

where

t = time segment for the early period

$p^2(t)$ = squared impulse response

Center time (T). T is the time (in milliseconds) it takes to reach the center of gravity of integrated energy level vs. time at a given location in a room. It is highly correlated to EDT and hence to RT. This measure is used to avoid the sharp cutoff points used in the Early/Late Energy Ratio. This parameter was proposed by Cremer (1978) and contributes to the subjective perception of "clarity."

The quantitative measure is:

$$T = \int_0^{\infty} t \cdot p^2(t) dt / \int_0^{\infty} p^2(t) dt \text{ (Bradley, 1990)}$$

where

t = reverberant decay period

$p^2(t)$ = squared impulse response

Lateral energy fraction (LEF) and spatial impression (SI). The LEF at a particular location is the ratio in dB of the early lateral reflected energy received (measured for a time interval starting at 5 ms after the sound impulse to 80 ms after) to the total early energy received at that location (direct plus early reflected energy) measured for a time interval of 80 ms after the sound impulse. The SI is a measure of the degree of envelopment or the degree to which a listener feels immersed in the sound, as opposed to receiving it directly. It is linearly related to the LEF and an equation has been derived for SI based on the LEF by Barron and Marshall (1981). These parameters contribute to the subjective perception of "envelopment," "spaciousness," "width of the sound source" and "spatial responsiveness/impression." The quantitative measure for LEF is:

$$LEF = \frac{\sum_{5ms}^{80ms} r \cdot \cos\phi}{\sum_{0ms}^{80ms} r} \text{ (Stettner, 1989)}$$

where

r = reflection energy of each ray

ϕ = angle in the horizontal plane that the reflected ray makes with an axis through the receiver's ears

$$SI = 14.5 (LEF - 0.05) \text{ (Barron \& Marshall, 1981)}$$

A modified measure of SI to include loudness is:

$$SI = 14.5 (LEF - 0.05) + (L - L_0) / 4.5 \text{ (Barron \& Marshall, 1981)}$$

where

L_0 = threshold loudness for spatial impression

L = loudness

LEF is related to width of the room according to Gade (1989):

$$LEF = 0.47 - 0.0086 * W \text{ where } W \text{ is the width.}$$

Bass ratio (BR), bass level balance (BLB), early decay time ratio (EDTR) and center time ratio (CTR). These parameters are single number parameters (ratios) related to the relative strength of the bass sound to the mid frequency sounds. The BR is based on RT and was developed by Beranek (1962). When the ratio is based on L, it is called the BLB. When the ratio is based on the EDT, it is called the EDTR. When it is based on T, it is called the TR. Measures have been developed for all these parameters. The BR, BLB, EDTR and TR contribute to the subjective perception of "tonal color" or "tonal balance." The quantitative measurements for the above are:

$$BR = (RT_{125Hz} + RT_{250Hz}) / (RT_{500Hz} + RT_{1kHz}) \text{ (Gade, 1989)}$$

$$BLB = (L_{125Hz} + L_{250Hz} - L_{500Hz} - L_{1kHz}) / 2 \text{ (Gade, 1989)}$$

$$EDTR = EDT_b / EDT_m \text{ (Barron, 1988)}$$

$$TR = T_b / T_m \text{ (Barron, 1988)}$$

where

b = bass frequency

m = mid frequency

Hz = Hertz (cycles/second)

k = 1000

Useful to detrimental ratio (U), speech transmission index (STI) and rapid speech transmission index (RASTI). The U parameter was developed by Lochner and Burger (1958, 1964). It is the ratio in dB of the useful early energy received at a certain location to the detrimental energy constituted by the sum of the energy of the later arriving sound and the ambient or background noise energy. The U parameter of Lochner and Burger was further simplified by Bradley (1986b, 1986c). The U parameter is measured for time intervals that constitute the "early" period, which is usually 50 ms or 80 ms. This parameter contributes to speech intelligibility in rooms. The quantitative measure for U is:

$$U_t = 10 \log [\int_0^t p^2(t) dt / (\int_t^\infty p^2(t) dt + \text{ambient energy})]$$

(Bradley, 1990)

where

t = time segment of the early period

$p^2(t)$ = squared impulse response

The STI and RASTI were developed by Houtgast and Steeneken (1973). They are measures for the intelligibility of speech in rooms. The acoustical properties of rooms and the ambient noise in rooms diminish the natural amplitude modulations of speech. The STI measure

assesses the modulation transfer functions (MTFs) for the 96 combinations of 6 speech frequency bands and 16 modulation frequency bands. From this matrix of values, a single value between 0 and 1.0 is derived using a system of weightage called the STI. STI has also been computed from the squared impulse response by Bradley (1986b) according to a method proposed by Schroeder (1981). Both STI and RASTI are strongly correlated to U values. A quantitative method for calculating the MTF from the squared impulse response is shown below:

$$\text{MTF}(\omega) = \int_0^\infty p^2(t) e^{-i\omega t} dt / \int_0^\infty p^2(t) dt \quad (\text{Schroeder, 1981})$$

where

$$\omega = 2\pi * \text{frequency}$$

$$p^2(t) = \text{squared impulse response}$$

Subjective Perceptions Related to Acoustical Parameters

The subjective perceptions related to the acoustical parameters and their references in the research literature are presented next. Because of the different semantic interpretations of subjective perceptions, it is a very difficult task to experimentally correlate acoustical parameters with subjective perceptions. Many of the linkages have been established based on intuition, experience and convention rather than by scientific methods. Experimental studies that record both subjective responses and objective measurements of acoustical parameters at each location in a room are needed to correlate these two factors. Very few such studies have been done so far. Factor analysis is another method to establish these correlations.

Studies that have established specific relations between the acoustical parameters and subjective perceptions are discussed next.

The relation between Reverberation Time and the perception of reverberance is intuitively obvious. Resonance, fullness and liveness (Beranek's definition) are synonymous with reverberance. The relationship of Early Decay Time to the perception of reverberance was first established by Schroeder (1965). In tests conducted by Barron (1988), a moderate correlation (correlation coefficient = 0.39) between Reverberation Time and the perception of reverberance was established. However, Barron found that the Early Decay Time had a stronger correlation with the perception of reverberance (correlation coefficient = 0.53). This supported Schroeder's work. Reverberation Time also correlated negatively with the perception of clarity (correlation coefficient = -0.51). Early Decay Time correlated negatively with the perception of clarity to a lesser degree (correlation coefficient = -0.33).

Barron also found that Loudness measured as the Total Sound Level and Early Sound Level strongly correlated with the subjective perception of loudness. The Strength Index computed from Sound Levels was shown to be strongly linked to the perception of loudness by Wilkens and Lehmann (reported in Cremer, 1978). Barron also found that these sound levels were correlated with the perception of intimacy. The sound levels also correlated with the perception of envelopment. The latter two correlations might be due to the latitude in the semantic interpretation of the subjective qualities, e.g., intimate meaning near, loudness suggesting near, loudness being overwhelming, overwhelming meaning envelopment, hence the correlations. Further, Barron found that the Early Decay Time Ratio and the Center Time Ratio were moderately correlated with the perception of tonal balance (correlation coefficient

= 0.35). A stronger relation between them was established by Wilkens and Lehmann (reported in Cremer, 1978). Barron also found that the Lateral Energy Fraction correlated moderately with the perception of envelopment (correlation coefficient = 0.30). Lehmann and Wilkens (1980) found correlations between Total Sound Level and the perception of loudness, Center Time and the perception of clarity (a negative correlation), and Early Decay Time and the perception of reverberance.

The relationship between Lateral Energy Fraction and the perception of spatiality was established by Barron and Marshall (1981). They also developed the Spatial Impression parameter which is derived from the Lateral Energy Fraction and is more strongly related to the perception of spatiality. This relationship was refined by Blauert (1986). Nuances in the interpretation of the Lateral Energy Fraction and its relationship to spatiality were established by Keet, Kuhl, Reichardt and Schmidt (reported in Cremer, 1978). The relationship between the Inter-Aural Correlation Coefficient and the perception of spatiality, which was perceived as the angle of the reflected sound from the median plane and the width of the hall, was established by Ando (1985).

The relationship between the Initial Time Delay Gap and intimacy was suggested by Beranek (1962). He also suggested the relation of the Bass Ratio to the subjective perception of warmth. Hawkes and Douglas (1971) found that the Initial Time Delay Gap was correlated to the perception of intimacy. The relationship between the Early/Late Energy Ratio and the perception of musical clarity was established by Reichardt (1981) and Eysholdt (1975). The relationship between Late/Early Energy Ratio and running liveness was established by Schultz (1965). Liveness was first related to the Late/Early Energy Ratio by Maxfield and Albersheim

(1947). Beranek and Schultz (reported in Cremer, 1978) proposed a 50 ms time interval to compute the early part of the energy in the Late/Early Energy Ratio.

The relationship of the Useful/Detrimental Energy Ratio to the intelligible perception of speech was established by Lochner and Burger (1964). The ratio was further simplified by Bradley (1986b). The relationship of the Speech Transmission Index and the Rapid Speech Transmission Index to the intelligible perception of speech was established by Houtgast and Steeneken (1973 and 1980). The relationship of the Early/Late Energy Ratio to distinctness (Deutlichkeit) or definition was established by Thiele (1953). This relationship was based on the human ear's ability to integrate the direct sound and early reflections and perceive it as different from the later arriving sound. Finally, the initial time delay gap is related to clarity because an early reflection reinforces the direct sound and makes it sound clearer and louder. A time delay of around 50 ms causes the direct sound to blend with the reflected sound. This is called "the limit of perceptibility" and is caused by the inertia of our hearing system. This was demonstrated by Haas and is called the Haas effect (1972). The subjective perception characteristics related to each of the acoustical parameters are shown below (in parentheses). The list reflects only positive correlates of the acoustical parameters.

1. Reverberation Time (reverberance, resonance)
Early Decay Time (fullness, liveness)
2. Room Constant (reverberance, loudness)
3. Overall Loudness or Strength of Sound Source (loudness)
4. Initial Time Delay Gap (intimacy, clarity)
5. Early/Total Energy Ratio (distinctness, definition)

Early/Late Energy Ratio (clarity)

Late/Early Energy Ratio (running liveness)

6. Useful/Detrimental Ratio (speech intelligibility)

Speech Transmission Index & Rapid Speech Transmission Index

(speech intelligibility)

7. Bass Ratio (tonal color)

Bass Level Balance (tonal balance)

Treble Ratio (tonal color)

Early Decay Time Ratio (balance between clarity and reverberance)

Center Time & Center Time Ratio (balance)

8. Lateral Energy Fraction (spatial envelopment)

Spatial Impression (spatial responsiveness, width of sound source)

The different acoustical parameters cited above can be resolved into related groups that have corresponding subjective perception characteristics. The parameters in items 1 and 2 (group 1) reflect the perception of reverberance, resonance, fullness and liveness all of which are related. The parameter in item 3 (group 2) reflects the perception of loudness. The parameters in items 4, 5 and 6 (group 3) reflect the perception of clarity, distinctness, definition and intelligibility all of which are related. The parameters in item 7 (group 4) reflect the perception of different kinds of balance. The parameters in item 8 (group 5) reflect the perception of spaciousness and envelopment. These groups of subjective perception characteristics can be classified as follows:

1. Reverberance

2. Loudness
3. Clarity
4. Balance
5. Spatiality/Envelopment

A similar grouping was derived by Bradley (1990). Bradley found these subjective perceptions to be linked to simple energy summations over different time intervals and their ratios as well as the rate of decay of the energy. Similar groupings have also resulted from factor analyses done by Gottlob, Wilkens, Lehmann, Eysholdt, Yamaguchi and Siebrasse (reported in Cremer, 1978).

Selection of Acoustical Parameters

Five characteristics were identified as significant subjective perception factors for the determination of overall acoustical quality. They were reverberance, loudness, clarity, balance and envelopment. Parameters responsible for those subjective perceptions were incorporated in a system (both statistical and analytical) that derived the spatial parameters of the auditorium from the acoustical parameters. It must be remembered that, in the generation stage, acoustical parameters were not the only factors determining the spatial form of the auditorium. Other factors like seating requirements, visual constraints and other programmatic requirements along with the acoustical parameters determined the spatial form of the auditorium. Where the effects of the parameters intersected, simple optimization techniques were used to resolve the situation. These included averages, maxima and minima. In future implementations, more complex optimization techniques are planned to be used.

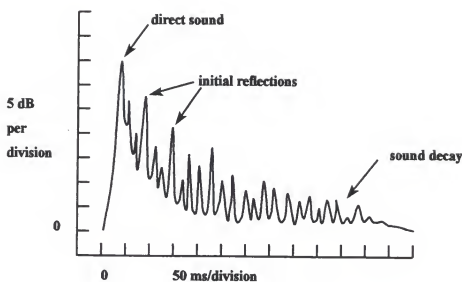


Figure 19. Energy impulse response graph (adapted from Siebein, 1989).

Based on studies done so far, a generative system based on macrostatic statistical relationships and some analytical theory has been developed by the author. A macrostatic study of the variation of sound energy at a location in the auditorium (the variation is reflected in the integrated energy in the impulse response) involves examining the relationships of the acoustical parameters (which are derived from the energy impulse response graph) as aggregate measurements and relating them to architectural parameters. This is opposed to the microdynamic interpretation of sound energy variation at a location which requires an analytical model. An example of an energy impulse response graph is shown in Figure 19. Information from this energy impulse response graph is transformed into the spatial form of the auditorium through acoustic sculpting. This makes acoustic sculpting a process of graphic

transformation. The generative system is described next. The values of the acoustical parameters for use in the generative system are to be drawn from a database of acoustical measures in different architectural settings that have been subjectively evaluated as desirable.

The Generative System

The generative system used to create the spatial design of the auditorium is based on relationships between spatial parameters and acoustical, functional and programmatic parameters. These relationships are based on the work of various researchers and are used to transform the acoustical, functional and programmatic parameters into spatial parameters. The acoustical, functional and programmatic parameters (independent variables) can be manipulated in the system at any time and in any order. They are on equal footing in terms of the order of manipulation. Consequently, the design process for creating the spatial design of the auditorium can begin with the setting of any parameter. For example, the performance mode of the auditorium is selected from a pop-up menu that appears when you click in the performance mode box with the menu button on the mouse. Five choices are presented to the user. They are:

1. Theater
2. Drama
3. Musical
4. Symphony
5. Opera

Based on the user's choice, the proscenium dimensions are set according to the performance mode. From the proscenium dimensions, the width of the stage, the height of the stage and the depth of the stage are set. These settings are based on recommendations in the Architectural Graphic Standards edited by Ramsey and Sleeper (1993).

The depth of the stage apron is set using a slider that allows the user to select a value from 5 feet to 20 feet. The stage platform height is set at the maximum value recommended in the Architectural Graphic Standards (Ramsey and Sleeper, 1993). The first row distance from the edge of the stage apron is decided by the visual requirement that a human figure subtend an angle of 30 degrees at the first row (Ramsey & Sleeper, 1993). This dimension is added to the stage apron depth to give the distance of the first row from the sound source. The maximum distance allowable in the auditorium from the acoustical consideration of loudness is calculated from the relation that follows, which is based on an average of statistical relations found in the research of Hook (1989) and Barron (1988):

$$D = \text{dB (decibels)} / 0.049 \text{ (feet)}$$

where

D = maximum distance allowable based on dB loss.

dB = the dB loss allowable.

The desired loudness loss from the initial loudness of the sound source is selected for the receiving location using a slider that allows the user to choose a value from 3 dB to 8 dB. The lower limit of 3 dB was chosen because the human ear begins to perceive a drop in loudness for a drop of about that loudness level. A loudness loss of 6 dB results from the doubling of distance from the source.

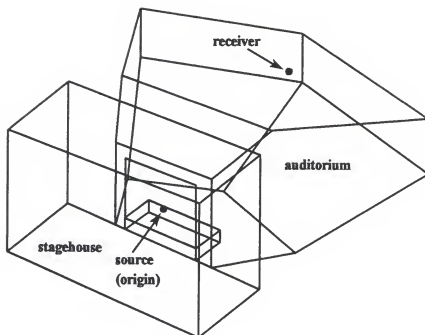


Figure 20. Model of the proscenium-type auditorium.

The reference point for all the dimensional variation of the auditorium is a point above the stage at the middle of the proscenium and at a height of 5.5 feet. The height of 5.5 feet represents the height of the eyes and ears of an average human being from the ground. This point is also the origin for the viewing system of the auditorium. The main receiving location that determines the spatial form of the auditorium is a point at the rear of the auditorium that is in direct line with the sound source and perpendicular to the proscenium plane (see Figure 20). The maximum distance from the loudness criterion is compared to the maximum distance set by the visual clarity criterion. The minimum of the two distances is set as the maximum distance from the source allowable in the auditorium.

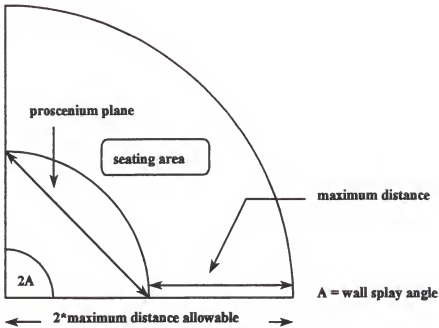


Figure 21. Determination of the wall splay angle from the seating area.

The capacity of the auditorium is obtained from the user also using a slider. The slider allows the user to select a value for the capacity that ranges from 500 to 3000. The area per seat is also input using a slider with a value range from 4 to 8 square feet. Using the input for the capacity and the area per seat in the auditorium, the total seating area along with the area of the aisles is calculated. This area is considered as a portion of a circular sector starting at the proscenium with a radius that is twice the maximum distance. Figure 21 illustrates this aspect.

The total seating area is also multiplied by the average height of the auditorium to arrive at the volume of the auditorium. This volume, along with a user supplied reverberation

time (an average of the reverberation time at 500 Hz and 1000 Hz), is used in Sabine's formula (1964) to calculate the Room Constant of the auditorium to achieve the specified reverberation time. The absorption due to the audience (using a 50% occupancy rate) and the absorption due to the air is taken into account in calculating the Room Constant. Mean absorption coefficients for the wall and roof surfaces and the wall surfaces alone are calculated and presented to the user as a recommendation. These absorption coefficients will dictate the materials to be used in the construction of the interior of the auditorium. According to Sabine's formula,

$$RT = (0.049V / S_T * \alpha)$$

where

RT = reverberation time

V = volume of the room in ft^3 ;

S_T = total surface area of room in ft^2 ;

α = mean absorption coefficient of the room.

The splay angle of the side walls from a line perpendicular to the proscenium is then calculated from the following equation (see Figure 21 for the basis):

$$\alpha(\text{angle}) = (60 * \text{total seating area}) / (\pi * \text{maximum distance}^2)$$

This angle (α) is compared to the angle set by visual requirements, which is 30 degrees, and the angles set by the Inter Aural Cross Correlation and the Treble Ratio. An optimum wall splay angle (the minimum) is then derived from these measures. The splay angle is also set to start beyond the proscenium width by a nominal distance of 6 feet. This is for obvious visual access reasons.

The next thing computed is the slope of the seating. This is derived from the following equation found in Cremer (1978):

$$\alpha = \epsilon * \ln(D/D_0)$$

where

α = angle of floor slope

ϵ = arcTan of (source height-1.75 m.)/(distance of first row from source)

D = maximum distance or length of auditorium

D_0 = distance of first row from source

$\ln$ = natural logarithm

From this the maximum height of the sloped floor is calculated using simple trigonometry.

This sets the vertices and planes that represent the sloped floor of the auditorium.

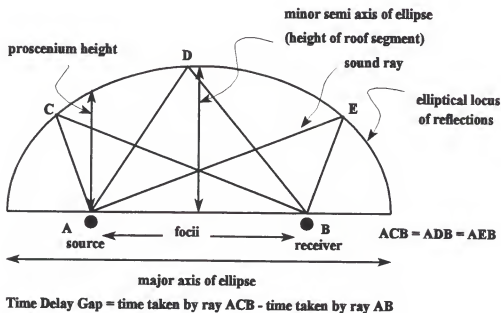


Figure 22. Elliptical field implied by reflected sound rays.

The coordinates of the roof segments are then calculated based on the elliptical fields implied by the Time Delay Gap (TDG) measurements (see Figure 22). This is based on the concept that the locus of the points generating reflected rays of an equal travel path from a source to a receiver is an ellipse. The TDG measurements at the main receiver location set the coordinates of the roof segments of the auditorium. Four TDG measurements representing four reflections are used to derive the coordinates of four roof segments of the auditorium. A fifth roof segment slopes from the fourth segment to the rear of the auditorium. The height of the first roof segment is set to be greater than the proscenium height. All the vertices and planes of the articulated roof are hereby set.

From this procedure, the heights of the roof segments of the auditorium based on the TDG measurements are determined. Using these, the average height of the auditorium is computed. The average height is used to calculate the volume of the auditorium. The height of the ceiling at the rear of the auditorium is set by adding a nominal height (9 feet) to the maximum height set by the floor slope. Balconies are automatically introduced in the auditorium model if the wall splay angle based on the seating area exceeds the visual constraint angle of 30 degrees. The seating area cut off by maintaining the visual constraint angle of 30 degrees is provided in the balcony. The clearance height of the balcony soffit is calculated with visual access to the proscenium in mind as well as the recommended value from Ramsey and Sleeper (1993). The slope of the balcony floor is maintained at the maximum allowable which is 30 degrees. The diagram identifying the parameters that define the auditorium with the balcony is shown in Figure 23.

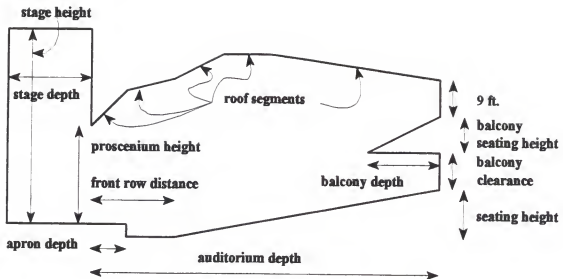


Figure 23. Section through the auditorium showing the different parameters.

The incorporation of adjacent lobby and lounge areas in the model has not been implemented at this stage of software development. However, it is a part of the next stage of software development. An interface is currently being developed that can transfer the computer model generated by this system in a format readily accepted by commercial CAD packages (DXF format) for design development. The complete computer code for this system is provided in Appendix B. A general description of the design systems implemented for the design of fan-shaped and rectangular proscenium-type auditoria is presented next. The details of the computer model are included in the chapter on results.

The Implemented Object-oriented Design Systems

For a first-hand experience in the creation of design systems using object-oriented computing, two design systems were developed for the preliminary spatial design of proscenium-type auditoria. The spatial forms of proscenium-type auditoria generated by the design systems are based on the concept of acoustic sculpting. The auditorium is modeled as a computational object. Various acoustical, functional and programmatic parameters are its data. Procedures that compute acoustical data, procedures that compute the spatial parameters of the auditorium and procedures that create the different graphic representations of the auditorium are its operations. The various parameters are interactively controlled to produce various designs of auditoria. The mechanism of inheritance is used to develop the second design system for the design of rectangular proscenium-type auditoria. This system is developed with minimal changes to the generative process in the first system. It is identical in function to the first system and has the same interface as the first system. The second system can be considered as a subtype of the first system. The same topology is maintained in the second system but the wall splay angles are forced to zero creating the rectangular proscenium-type auditorium. The wall splay angle generated by the computer model of the proscenium-type auditorium is used to determine the width of the rectangular proscenium-type auditorium. The width generated by the wall splay angle of the basic proscenium-type auditorium is added to the proscenium width to determine the width of the rectangular proscenium-type auditorium. The width generated by the wall splay angle is divided in half, and the two halves are added to each end of the proscenium.

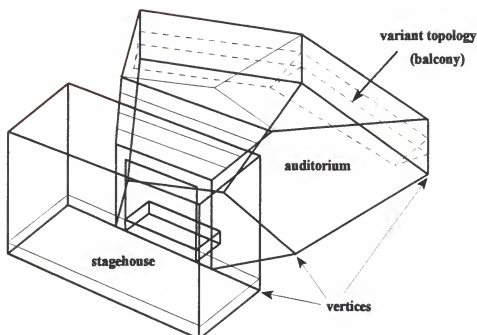


Figure 24. Topology of the proscenium-type auditorium.

The spatial design of the auditorium in both systems is based on constants, independent variables and derived variables. The independent variables are manipulated by using a graphic interface. These variables are used to generate sets of vertices and planes in three-dimensional space that are linked to form wire-frame and shaded plane images of the auditorium. The topology used to link the vertices and the planes is based on the spatial configuration of the proscenium-type auditorium. The topology sets the topology (see Figure 24). The topology that connects the vertices and planes is not fixed. It is a variant topology because balconies are introduced in the spatial design of the auditorium only when the

The generative system described in Chapter 2 is used to create interactive software developed with the Visual WorksTM object-oriented programming environment from ParcPlace Systems who are developers of SmalltalkTM products. The software uses the model-view-controller paradigm in the Smalltalk programming environment¹¹ and has a user-friendly graphic interface with which to input acoustical, functional and programmatic parameters. The model-view-controller is a framework (Wirfs-Brock & Johnson, 1990) of three computational objects which are the model, the view and the controller. A model is any computational object. In this case, it is the computational model of the auditorium. A view is an object that is a particular representation of the model. Many views can be linked to a single model to represent different aspects of the model. The views in the implemented systems are the spatial images of the auditorium, the values of the various parameters and the data report of the auditorium. The views that show the values of the different parameters are input boxes that have been set in the read mode. Each parameter view has a controller that allows interactive manipulation of the parameter. The controllers in the implemented systems are the pop-up menu associated with the performance mode parameter and sliders associated with each of the other parameter views. When the model is changed, the various views related to the model are updated. A model-view-controller system is used in this project to provide a dynamic design environment. In the systems, the models change instantly with changing input of the parameters. The images of the auditorium are depicted in true perspective. Once the models are generated, they can be viewed from any angle and from any distance by manipulating the parameters of distance, latitude and longitude of the eyepoint. The systems

¹¹The paradigm is described elaborately by Krasner and Pope (1988)

can be used to rapidly generate alternate designs based on the various parameters. A single light source has been added to the models to enhance the shaded plane image of the auditorium. The distance, latitude and longitude of the light source can also be manipulated using sliders like the other parameters.

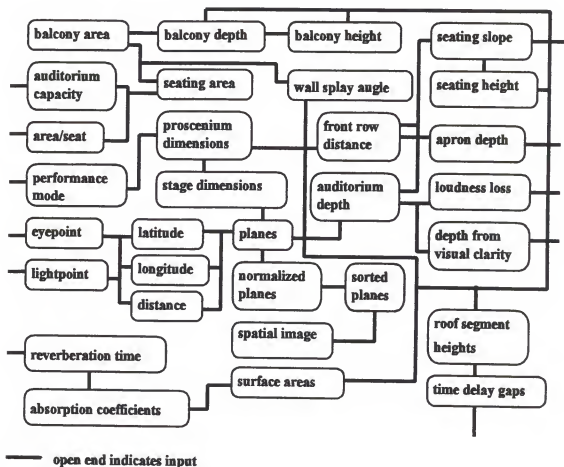


Figure 25. Relationships of key parameters in the auditorium model.

To limit the scope of the design to manageable limits, the initial versions of the two systems have only incorporated certain acoustical parameters associated with the spatial design of the auditorium. The total number of independent variables in each of the systems is twenty-one, indicating their complexity. There are nine derived variables and two variables that store the wire-frame and shaded plane views. Numerous intermediate parameters are computed during the process of determining the spatial parameters of the auditorium. The computer code in Appendix B is complete and includes both models.

These design systems use a reversal of the process of acoustical simulation achieved by Stettner (1989). In Stettner's work, acoustical parameters of computer models of spaces are derived through a simulation of sound propagation. A diagram showing the relationships among the different variables of the generative system is shown in Figure 25. Note the similarity of this diagram to the layout of an integrated circuit. This reinforces the concept discussed in Chapter 1 that an object-oriented system is a software integrated circuit. The design systems are run on a desktop computer using the Windows™ operating system and the Visual Works™ programming environment.

CHAPTER 3 RESULTS

The Computer Model of the Auditorium

The auditorium was modeled as a computational object. The following data and operations were defined for the auditorium object. In the Visual Works™ environment, data are called instance variables and operations are called methods. The naming convention used in this section is the Visual Works™ convention. The terminology in this section can be related directly to the computer code in Appendix B.

Instance Variables

The instance variables defined for the auditorium object are grouped into the following categories.

Viewing parameters

The viewing parameters are the following:

1. Eyepoint
2. Lightpoint
3. EyepointLatitude
4. EyepointLongitude
5. EyepointDistance

6. LightpointLatitude
7. LightpointLongitude
8. LightpointDistance
9. ViewingPlaneDistance

The viewing parameters contain data that are required to simulate the viewing of the auditorium from different viewpoints using different positions for a single light source. The eyepoint and lightpoint are defined in a spherical coordinate system. The location of the eyepoint and lightpoint are expressed in polar coordinates. The latitude, longitude and distance of the eyepoint and the lightpoint are the variables that are manipulated.

Stage parameters

The stage parameters are the following:

1. StageDepth
2. StageWidth
3. StageHeight
4. ProsceniumWidth
5. ProsceniumHeight
6. ApronDepth

The stage parameters contain data that are required to compute the physical dimensions of the stagehouse and the stage.

Auditorium parameters

The auditorium parameters are the following:

1. AuditoriumDepthFromVisualClarity

2. AuditoriumCapacity
3. AreaPerSeat
4. PerformanceMode
5. SeatingSlope

The auditorium parameters contain data that are required to determine the physical dimensions of the auditorium.

Acoustic parameters

The acoustic parameters are the following:

1. TimeDelay1
2. TimeDelay2
3. TimeDelay3
4. TimeDelay4
5. ReverberationTime
6. LoudnessLossAllowable
7. InterAuralCrossCorrelation (IACC)
8. Treble Ratio

The acoustical parameters contain data that are transformed using acoustic sculpting to yield the physical dimensions and the spatial parameters of the auditorium.

View parameters

The view parameters are the following:

1. Planes
2. PlaneView

3. FrameView

4. DataReport

The view parameters contain data that display the different views of the auditorium including the graphic views of the wire frame and shaded planar images of the auditorium. A text view of other data pertaining to the auditorium is stored as the dataReport parameter. These views represent different aspects of the auditorium model and present different kinds of information about the model. When the model is transformed, these views are updated to reflect the current state of the aspects they present. The concept of using multiple views in the computer model is based on ideas in database design where different views of data are presented in different formats.

These views could also have been used to show the different projections that architects normally use when they present designs, namely, sections and elevations. This option was not pursued, but could be easily implemented with the vertex data generated by the system. Procedures to generate the vertex data are part of the computer code in Appendix B. The vertex data consist of the coordinates of each vertex in three-dimensional Cartesian space. Combined with the planar data, these vertex data could be used to develop a ray tracing model of sound propagation in the computer model of the auditorium. This could then be used to derive an energy impulse response graph for locations in the auditorium. These graphs could then be convolved with sound signals to project how the auditorium would sound if built. The sound produced by the convolving process could, in effect, become another "view" of the auditorium, albeit an auditory one.

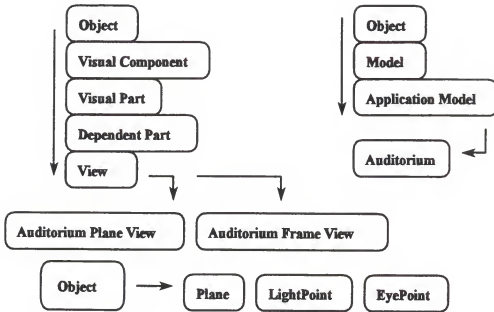


Figure 26. Class hierarchies of computational objects in the system.

The eyepoint and lightpoint are themselves objects. The planes used in the auditorium model are also objects. The planeView and the frameView instance variables hold AuditoriumPlaneView and AuditoriumFrameView objects. The class hierarchies of all these objects is shown in Figure 26. The eyepoint and lightpoint objects have the following instance variables:

1. Latitude
2. Longitude
3. Distance

The operations of the eyepoint and lightpoint objects relate to the setting and accessing of these instance variables.

The operations of the eyepoint and lightpoint objects relate to the setting and accessing of these instance variables.

The plane object represents the planes that make up the spatial model of the auditorium. The plane object has the following instance variables:

1. Color
2. Distance
3. ID
4. Points
5. XNormal
6. YNormal
7. ZNormal

The plane object through its operations can compute its color based on the eyepoint and lightpoint. It can transform itself into screen coordinates based on a transformation matrix generated by the eyepoint. It can compute its x, y and z normals, and its maximum and minimum z coordinates after being transformed into screen coordinates. It can also compute its distance from the origin. It can also set and access its instance variables. The delegation of all these operations to the plane object makes it possible for the auditorium to be defined as a set of planes without worrying about the operations required for their graphic representation. Operations in the plane objects can also be defined to compute the surface area of the plane. The surface areas of the planes can be used for the accurate calculation of absorption levels in the spatial model of the auditorium. The plane objects can also be used in a ray tracing model to simulate the propagation of sound in the auditorium.

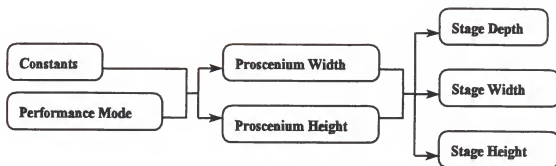


Figure 27. Relationship of performance, proscenium and stage parameters.

The AuditoriumPlaneView object displays a shaded planar view of the auditorium and the AuditoriumFrameView object displays a wireframe view of the auditorium. These objects automatically update their display when any of the auditorium's parameters are changed. The auditorium object also uses points which are PointVector objects and transformation matrices which are TransMatrix objects. Both these are subclasses of the Array object. The PointVector object is an array of four elements which are the x, y and z coordinates and the normalizing constant 1. The TransMatrix object is an array that contains a transformation matrix. This transformation matrix is generated using the latitude, longitude and distance of the eyepoint.

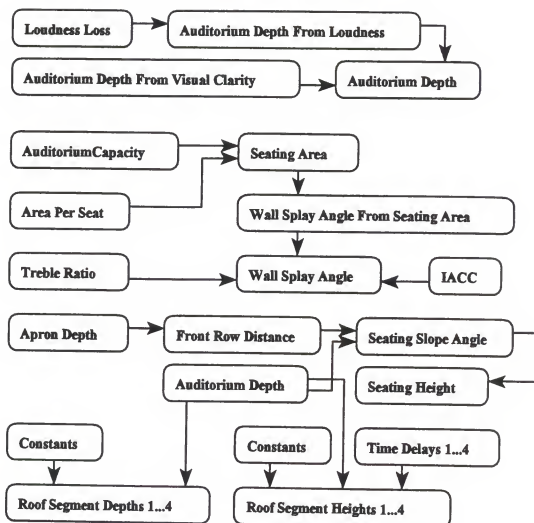


Figure 28. Relationship of input parameters.

Methods

The operations of the auditorium object are called methods in Visual Works™. These methods are grouped into protocols. The methods in each protocol are presented next. Methods that apply to an instance of the auditorium object are called instance methods. Methods that apply to the Auditorium class are called class methods.

Setting methods

The following methods are used to set the proscenium height and width:

1. ProsceniumHeight: aHeight
2. ProsceniumWidth: aWidth

The following methods are used to set the dimensions of the stage:

1. StageDepth: aDepth
2. StageHeight: aHeight
3. StageWidth: aWidth

The following methods are used to set the performance mode of the auditorium:

1. SetDrama
2. SetTheater
3. SetSymphony
4. SetMusical
5. SetOpera

The following methods are used to set the proscenium dimensions based on the performance mode and the stage dimensions based on the proscenium dimensions:

1. SetProsceniumDimensions
2. SetStageDimensions

The following methods set the eyepoint and lightpoint of the auditorium based on the latitude, longitude and distance specified for each of them:

1. SetEyepoint
2. SetLightpoint

The following methods set the successive time delays for the reflected sound based on user input:

1. SetTimeDelay1
2. SetTimeDelay2
3. SetTimeDelay3
4. SetTimeDelay4

The following methods compute the stage dimensions, planes and data report of the auditorium:

1. SetPlanes
2. SetStageDimensionsAndPlanes
3. SetStageDimensionsReportAndPlanes
4. SetDataReportAndPlanes

Accessing methods

The following methods access the different stage and proscenium dimensions that have been set:

1. ProsceniumWidth
2. ProsceniumHeight
3. StageDepth
4. StageWidth
5. StageHeight

Figure 27 shows the linkages between these methods.

The following methods calculate the different spatial parameters of the auditorium:

1. AuditoriumDepth
2. AuditoriumDepthFromLoudness
3. FrontRowDistance
4. SeatingArea
5. SeatingSlopeAngle
6. SeatingHeight
7. WallSplayAngleFromSeatingArea
8. WallSplayAngle
9. RoofSegmentDepth1.....RoofSegmentDepth4
10. RoofSegmentHeight1.....RoofSegmentHeight4

Figure 28 shows the linkages of these methods.

The following methods calculate the different spatial parameters of the balcony in the auditorium:

1. BalconyArea
2. BalconyDepth
3. BalconySeatingHeight
4. BalconyClearanceHeight

Figure 29 shows the linkages between these methods.

The methods that compute the balcony parameters are the ones that change from the first system to the second. This is because the forcing of the wall splay angle to zero in the second system makes it necessary to compute the balcony parameters in a different way. The balcony is no longer spread along the arc of a circle, hence the difference. The rectangular

configuration of the balcony in the second system also makes it necessary to adjust the constraints for the balcony parameters.

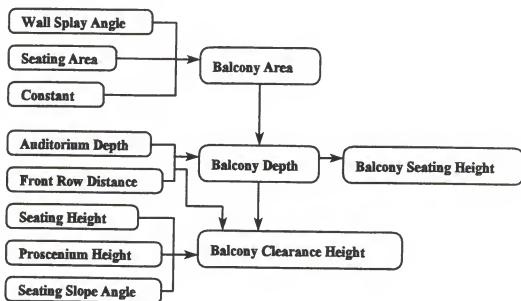


Figure 29. Relationship of parameters that define the balcony.

The following methods calculate the spatial and acoustical properties of the auditorium:

1. AverageAuditoriumHeight
2. AuditoriumVolume
3. ApproximateWallAndSurfaceArea
4. RoomConstant
5. AverageAbsorptionCoefficient
6. AverageWallAbsorptionCoefficient

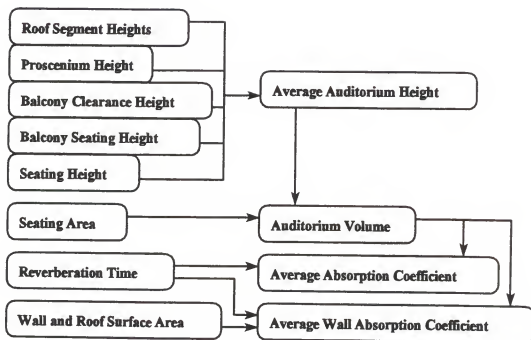


Figure 30. Relationships to compute acoustical parameters.

Figure 30 shows the linkages between these methods. The following methods are used to access the eyepoint, lightpoint, planeView, frameView and dataReport variables in the auditorium model:

1. Eyepoint
2. Lightpoint
3. PlaneView
4. FrameView
5. DataReport

The following methods are used to calculate the planes and vertices of the auditorium:

1. plane1.....plane32

2. v1.....v55

Initializing methods

The following method is used to initialize the parameters with a default value:

1. Initialize

Computing methods

The following method is used to compute the screen coordinate of a point defined in world coordinates of x, y and z using a viewing transformation matrix:

1. ComputeScreenCoordinates

Planes processing methods

The following methods are used to calculate the properties of the planes of the auditorium and sort them for display:

1. SetColoredPlanes
2. SetSortedPlanesNormalized

Aspect methods

The following methods are used to access the independent variables that the user supplies:

1. ApronDepth
2. AreaPerSeat
3. AuditoriumCapacity
4. AuditoriumDepthFromVisualClarity
5. EyepointLatitude
6. EyepointLongitude
7. EyepointDistance

8. LightpointLatitude
9. LightpointLongitude
10. LightpointDistance
11. ViewingPlaneDistance
12. SeatingSlope
13. PerformanceMode
14. LoudnessLossAllowable
15. ReverberationTime
16. TimeDelay1.....TimeDelay4
17. InterAuralCrossCorrelation (IACC)
18. Treble Ratio

Besides these methods, the auditorium object also has methods associated with its class that enable it to be incorporated into a computer-aided design system. These methods are used to create an instance of the auditorium object and also to define the methods used to create its graphic interface in the design system.

Class methods

1. New
2. WindowSpec
3. ModeMenu

These data and operations were used to develop the computer model of the auditorium. Together they define the auditorium computational object.

Results Achieved Using the Design Systems

To test the design systems and see if they produced results comparable to existing auditoria, parameters from two concert halls were used in the design systems. The two halls were the Boston Symphony Hall and the Kleinhans Hall in Buffalo, New York. The Boston Symphony Hall was chosen because it was rectangular in shape and its parameters could be used to test the design system for rectangular proscenium-type auditoria. The Kleinhans hall was chosen because it was a fan-shaped, proscenium-type hall. The parameters of the Kleinhans Hall were used to test the basic design system for proscenium-type auditoria. The input parameters were taken from Table B-1 and Table 2-2 in Chiang's dissertation (Chiang, 1994) or measured from scale drawings of the halls. The following parameters were identified and used based on the Boston Symphony Hall:

Auditorium Capacity: 2555

Area/Seat: 5.4 sq. ft..

Apron Depth: 6.5 ft.

Depth for Visual Clarity: 132 ft.

Seating Slope: 2.5 degrees.

Loudness Loss Allowable: 6.5 dB

Time Delay 1: 0.026 sec.

Time Delay 2: 0.028 sec.

Time Delay 3: 0.03 sec.

Time Delay 4: 0.032 sec.

Inter Aural Cross Correlation: 0.23

Treble Ratio: 0.99

Reverberation Time: 2.4 sec.

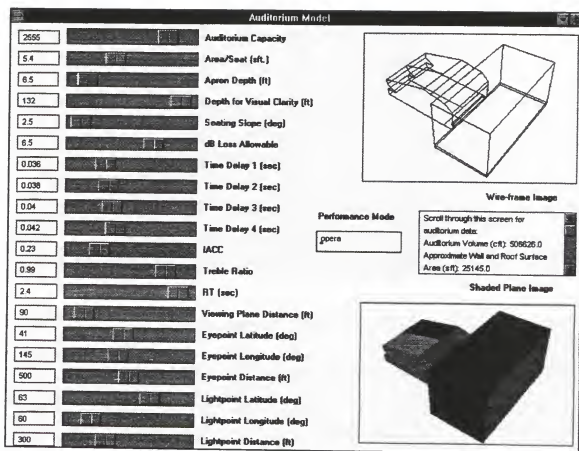


Figure 31. Printout of the computer screen showing the result produced by the design system for rectangular proscenium-type auditoria using the Boston Symphony Hall parameters.

The result produced by the design system is shown in Figure 31. The design system calculates the architectural and acoustical parameters for the design it produces. The following values were calculated by the design system for architectural and acoustical parameters:

Auditorium Volume: 506,626 cu. ft.

Approximate Wall and Roof Surface Area: 25145 sq. ft..

Seating Area: 13,797 sq. ft.

Average Height: 41.54 ft.

Average Width: 80.76 ft.

Auditorium Depth: 132 ft.

Room Constant: 3129.9

Average Absorption Coefficient: 0.12

Average Wall Absorption Coefficient: 0.245

The comparison of these results to the original values is shown in Figure 32.

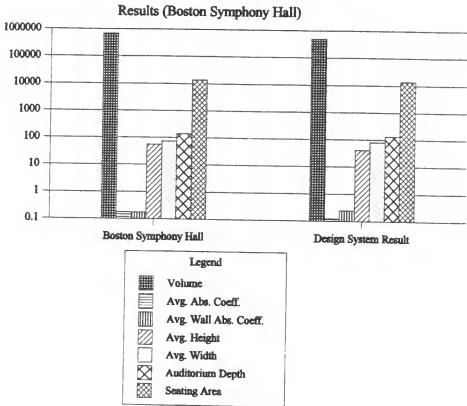


Figure 32. Comparison of the results produced by the design system for rectangular proscenium-type auditoria using the Boston Symphony Hall parameters.

In the second test, the following parameters were identified and used based on the

Kleinhans Hall:

Auditorium Capacity: 2839

Area/Seat: 7.2 sq. ft..

Apron Depth: 20 ft.

Depth for Visual Clarity: 105 ft.

Seating Slope: 5.0 degrees.

Loudness Loss Allowable: 5 dB

Time Delay 1: 0.02 sec.

Time Delay 2: 0.022 sec.

Time Delay 3: 0.024 sec.

Time Delay 4: 0.026 sec.

Inter Aural Cross Correlation: 0.51

Treble Ratio: 0.85

Reverberation Time: 1.6 sec.

Some of these parameters were taken from Table B-1 and Table B-2 in Chiang's dissertation (1994). Other parameters were measured from drawings of Kleinhans Hall that are part of the collection of the University of Florida research team on architectural acoustics. The Inter Aural Cross Correlation parameter was modified to approximate the average width of the Kleinhans Hall. If the original parameter had been used, only the width at the centroid of the hall would have been obtained.

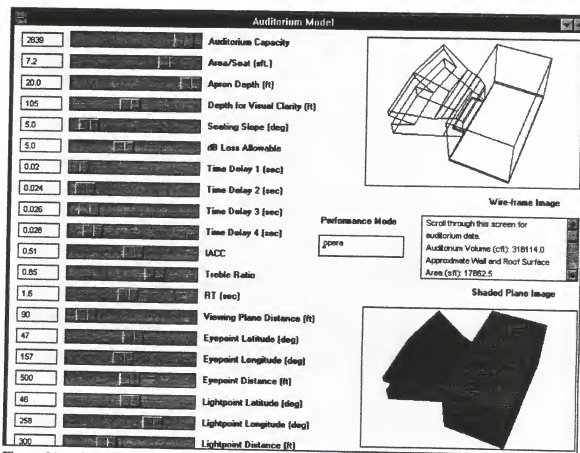


Figure 33. Printout of computer screen showing the result produced by the design system for proscenium-type auditoria using the Kleinhans Hall parameters.

The result produced by the design system is shown in Figure 33. The design system calculates the architectural and acoustical parameters for the design it produces. The following values were calculated by the design system for architectural and acoustical parameters:

Auditorium Volume: 318,114 cu. ft.

Approximate Wall and Roof Surface Area: 17662.5 sq. ft.

Seating Area: 20440.8 sq. ft.

Average Height: 38.97 ft.

Average Width: 123.33 ft.

Auditorium Depth: 102.04 ft.

Room Constant: 4812.29

Average Absorption Coefficient: 0.268

Average Wall Absorption Coefficient: 0.517

Wall Splay Angle: 23.9 degrees

The comparison of these results to the original values is shown in Figure 34.

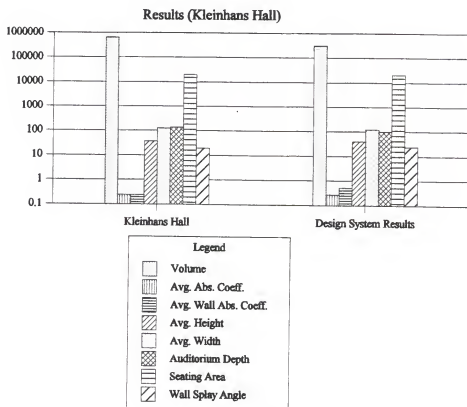


Figure 34. Comparison of results produced by the design system for proscenium-type auditoria using the Kleinhans Hall parameters.

The comparison charts clearly indicate that the design systems are reasonably close in terms of the acoustical and architectural parameters that they generate. The significant differences are in the volume of the two halls. The design systems consistently create designs with lesser volume. In the case of the Boston Symphony Hall, this is because a second level balcony was not included in the topological model of the auditorium in the design system. The average height of the design based on the Boston Symphony Hall is also less because of this reason. In the design based on the Boston Symphony Hall, all the seating area is accommodated on the ground floor and one balcony. In the Kleinhans Hall, there is a shortfall of 30.55% in the balcony area. In the original hall, the balcony area extends beyond the rear wall of the ground floor. This extension is not possible with the topological model in the design system. Hence, the volume is lesser in the design based on the Kleinhans Hall. In both the designs, the proscenium parameters could not be controlled because they are preset based on the performance modes. The performance mode of opera was used in both tests to approximate the original proscenium widths in the halls. The dimensions of the stagehouse were also preset in the design systems based on architectural standards (Ramsey & Sleeper, 1993). This resulted in stagehouses of large volumes that were attached to the auditoria. In future versions of these design systems, it might be more useful to independently control the proscenium width, the proscenium height and the stagehouse dimensions to account for variations.

In the design based on the Boston Symphony Hall, the average absorption coefficient is less (0.12 compared to 0.17). This figure can be attributed to the lesser surface area due to the lesser volume in the design. In the design based on the Kleinhans Hall, the average

absorption coefficient is a lot closer to the original value (0.27 compared to 0.24). The average height in the design based on the Kleinhans Hall is marginally more (38.97 ft. compared to 37.4 ft.), but the average height in the design based on the Boston Symphony Hall is significantly less (41.54 ft. compared to 55.6 ft.). This can be attributed to the absence of the second balcony in the design based on the Boston Symphony Hall. The wall splay angle in the design based on the Kleinhans Hall is also fairly close to the original value (23.9 deg. compared to 19.34 deg.). A higher than average Inter Aural Cross Correlation (IACC) parameter was used in the case of the Kleinhans Hall to obtain an average width closer to the original value (123.33 ft. compared to 127.4 ft.). The IACC parameter also influences the wall splay angle. A compromise value for the IACC parameter (0.51) had to be used to approximate the wall splay angle and the average width. This is not an unusual choice because the average IACC parameter in Kleinhans Hall is 0.34 representing the value at the approximate centroid of the fan-shaped auditorium. The average value of the IACC parameter is 2/3 the value of the parameter chosen which represents the value at the rear of the fan-shaped auditorium.

In both the tests, the design systems produced results that were comparable to the original auditoria. This was despite the limitation of mismatched topologies. The design systems were not intended to generate all existing auditoria in true detail. Even though replicating existing auditoria was not a major goal of the design systems, the systems produced designs that were reasonably close to the original auditoria whose parameters were used. Allowing for the limitations of the topological models in the design systems, the results produced for the main auditorium space were promising. This reinforces the claim that these

design systems are good preliminary spatial design tools for auditoria. However, the design systems have to be revised in order to accommodate variations in the designs that are of a practical nature. These include the independent control of seating slopes, proscenium dimensions and stagehouse dimensions. Since the design systems are preliminary design tools, there is the implicit understanding that the designs produced by these systems will be modified during the design development stage.

In order to test the effectiveness of the design systems when using parameters from auditoria of comparable topology, two additional tests were conducted using the parameters of the Music Hall at the Century II Center in Wichita, Kansas and the Theatre Maisonneuve in Montreal, Canada. In the first test, the following parameters were identified and used based on the Music Hall:

Auditorium Capacity: 2220

Area/Seat: 8 sq. ft..

Apron Depth: 20 ft.

Depth for Visual Clarity: 116 ft.

Seating Slope: 8 degrees.

Loudness Loss Allowable: 7.5 dB

Time Delay 1: 0.03 sec.

Time Delay 2: 0.032 sec.

Time Delay 3: 0.034 sec.

Time Delay 4: 0.036 sec.

Inter Aural Cross Correlation: 0.37

Treble Ratio: 0.43

Reverberation Time: 1.9 sec.

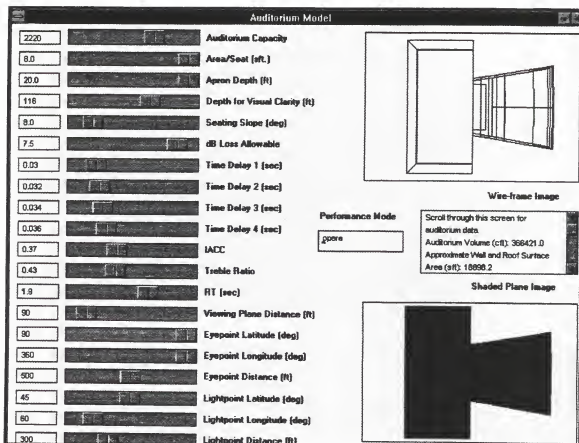


Figure 35. Printout of computer screen showing result produced by the design system for proscenium-type auditoria using the Music Hall parameters.

The result produced by the design system using the parameters of the Music Hall is shown in Figure 35, and the comparison of the original parameters to the parameters produced by the design system is shown in Figure 36. Comparisons were made only for the parameters of this hall that were available in the research literature surveyed (especially in Izenour, 1977). Whenever possible, parameters were measured from scale drawings of this hall found in Izenour (1977, p. 436).

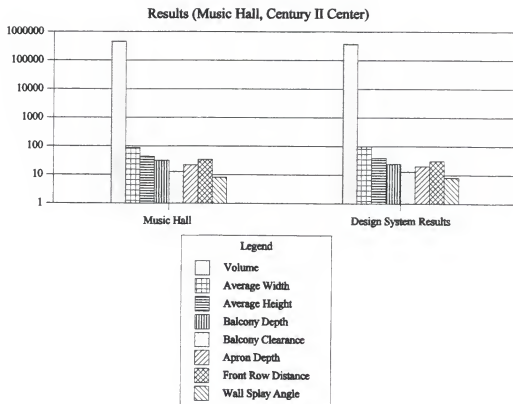


Figure 36. Comparison of results produced by the design system for proscenium-type auditoria using the Music Hall parameters.

The following values were calculated by the design system for architectural and acoustical parameters:

Auditorium Volume: 366,421 cu. ft.

Approximate Wall and Roof Surface Area: 18898.2 sq. ft..

Seating Area: 17760 sq. ft.

Average Height: 38.42 ft.

Average Width: 99.01 ft.

Auditorium Depth: 116 ft.

Room Constant: 3870.96

Average Absorption Coefficient: 0.2

Average Wall Absorption Coefficient: 0.408

Wall Splay Angle: 8.434 degrees

In the second test, the following parameters were identified and used based on the

Theatre Maisonneuve:

Auditorium Capacity: 1300

Area/Seat: 6.5 sq. ft..

Apron Depth: 5.5 ft.

Depth for Visual Clarity: 80 ft.

Seating Slope: 20 degrees.

Loudness Loss Allowable: 4 dB

Time Delay 1: 0.03 sec.

Time Delay 2: 0.032 sec.

Time Delay 3: 0.034 sec.

Time Delay 4: 0.036 sec.

Inter Aural Cross Correlation: 0.43

Treble Ratio: 0.51

Reverberation Time: 1.6 sec.

The actual Time Delay, Inter Aural Cross Correlation, Treble Ratio and Reverberation Time parameters were not available, so intuitive values were used for those parameters.

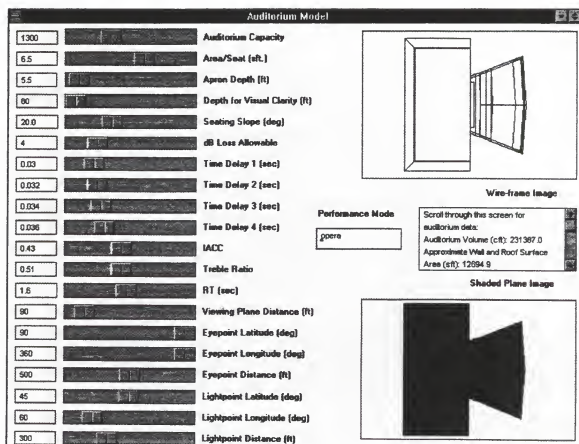


Figure 37. Printout of computer screen showing result produced by the design system for proscenium-type auditoria using the Theatre Misonneuve parameters.

The result produced by the design system using the parameters of the Theatre Misonneuve is shown in Figure 37, and the comparison of the original parameters to the parameters produced by the design system is shown in Figure 38. Comparisons were made only for the parameters of this hall that were available in the research literature surveyed (especially in Doelle, 1972). Whenever possible, parameters were measured from scale drawings of this hall found in Doelle (1972, p. 66). The seating slope angle and height parameters were not available for this hall. This is because the only scale drawing available of this hall were floor plans.

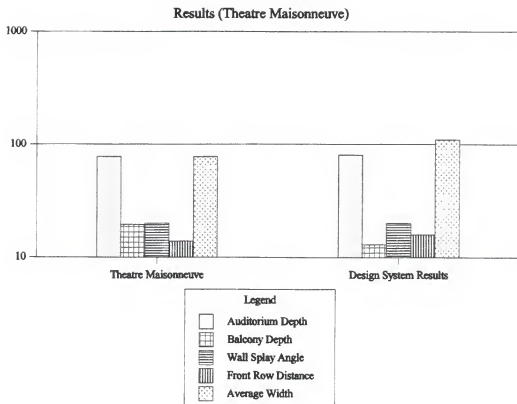


Figure 38. Comparison of results produced by the design system for proscenium-type auditoria using the Theatre Maisonneuve parameters.

The following values were calculated by the design system for architectural and acoustical parameters:

Auditorium Volume: 231,387 cu. ft.

Approximate Wall and Roof Surface Area: 12694.9 sq. ft..

Seating Area: 8450 sq. ft.

Average Height: 35.28 ft.

Average Width: 109.43 ft.

Auditorium Depth: 80 ft.

Room Constant: 3276.13

Average Absorption Coefficient: 0.253

Average Wall Absorption Coefficient: 0.528

Wall Splay Angle: 20.0521 degrees

In both these tests, the results are much closer to the original parameters than in the first two tests. In the Music Hall model, the following results were achieved showing the closeness of the results to the original parameters (the original values of the parameters are shown in parentheses):

Volume: 366,421 cu. ft. (460,000 cu. ft.)

Average Auditorium Height: 38.42 ft. (42.97 ft.)

Average Auditorium Width: 99.01 ft. (88 ft.)

Balcony Depth: 24.23 ft. (31.25 ft.)

Balcony Clearance Height: 12.75 ft. (12.5 ft.)

Front Row Distance: 30.39 ft. (34.38 ft.)

Wall Splay Angle: 8.094 degrees (8 degrees)

In the case of the Theatre Maisonneuve, the following comparative results were achieved (the original values of the parameters are shown in parentheses):

Balcony Depth: 12.99 ft. (19.44 ft.)

Front Row Distance: 15.89 ft. (13.89 ft.)

Wall Splay Angle: 20.052 degrees (20 degrees)

Average Auditorium Width: 109.43 ft. (77.78 ft.)

Validation of the Computer Model of the Auditorium

The spatial forms generated by the design systems combine the effects of more than one parameter. The auditorium capacity and area per seat are programmatic parameters. The proscenium and stage dimensions based on the performance mode are functional parameters. The depth for visual clarity and other visual clearances are visual parameters. The reverberation time, treble ratio, inter aural cross correlation, time delays of reflections and loudness loss are acoustical parameters. All these kinds of parameters combine to derive the spatial form of the auditorium. From the spatial form of the auditorium, it is possible to derive the original parameters by reversing the algorithms. For example, though the loudness loss is used to derive the depth of the auditorium, the auditorium depth has been optimized with the depth for visual clarity. This will make it difficult for the same loudness loss to be recorded in the resulting model at the receiver location. Since more than one kind of parameter has been optimized in the resulting model, the original parameters will not be reflected in the resulting model except in their optimized form. This can be verified using the spatial model generated by this system in an acoustical simulation software package like AcoustiCAD™. This kind of verification was not attempted as a part of this dissertation.

CHAPTER 4 DISCUSSION

A New Computable Model of Architectural Design

There are many advantages in using the object-oriented paradigm for the development of computer-based design systems in architecture. The main advantage of the object-oriented approach is a computational basis for the creation of new types of computer-based design systems in architecture. These systems are based on modeling architectural design as synthesizing interaction. The synthesizing interaction model has fundamentally different implications for the design of computer-based design systems in architecture. This model facilitates the creation of computer-based design systems that generate architectural designs by the dynamic synthesizing interaction of physical and conceptual entities that are modeled as computational objects (see Figure 39). It is more common for architectural designs to result from a dynamic synthesizing interaction of physical and conceptual entities than it is from an explicit problem solving process. Conventional computer-based systems that supposedly aid the architectural design process normally only provide a medium to represent physical architectural entities. These physical entities are complex topological constructs synthesized from primitive solid geometric entities or planar surfaces. Conceptual entities can be represented in conventional systems only if their representation is geometric. Normally, conceptual entities are not represented directly in conventional systems. The architectural

design process on conventional systems is a synthesis of physical architectural entities represented through Constructive Solid Geometry (CSG) or Boundary Representation (B-rep). Conceptual entities in the mind of the designer regulate the synthesis of the physical entities. There is no explicit representation of conceptual entities in the design process. Conceptual entities can only be inferred from the organization of the physical entities. Conceptual entities are not engaged or manipulated directly in conventional systems. This significant drawback can be overcome in computer-based design systems where conceptual entities are modeled along with physical entities. Conceptual entities actively engage physical entities in the synthesizing interaction model of the architectural design process.

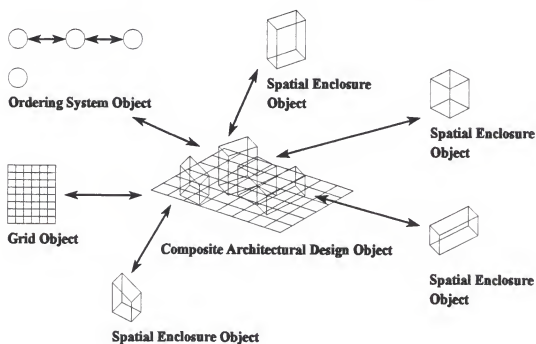


Figure 39. Architectural design as the synthesizing interaction of physical and conceptual entities modeled as computational objects.

Architectural design has been characterized in many different ways. In whatever way architectural design may be characterized, it involves the synthesis of physical and conceptual entities. Physical entities such as building components (materials and products) and conceptual entities such as architectural space, circulatory systems, structural systems and ordering systems are synthesized in architectural designs. These physical and conceptual entities can be modeled as computational objects in an object-oriented system. These objects can compute their spatial form akin to computing a shape based on design rules (as in shape grammar), display their image in different kinds of representations, provide context-based abstractions of themselves for analysis with different considerations and propagate changes to their different representations and abstractions when modified. Each of these objects will have a protocol for interaction with other objects. The definition of the interaction protocol for each architectural object becomes the main task of the designer of an object-oriented computer-based design system. Another important task is managing the synthetic object generated by the interaction of individual objects through an object-oriented database.

Architectural Entities as Computational Objects

Architectural entities are physical or conceptual. Physical architectural entities are individual building components (materials and products) and assemblies of building components that behave as individual components. For example, a brick is an individual component. A wall or arch made of bricks is an aggregate component whose behavior can be abstracted and modeled. Conceptual architectural entities are intangible entities like circulatory systems, ordering systems and structural systems.

A COLUMN OBJECT

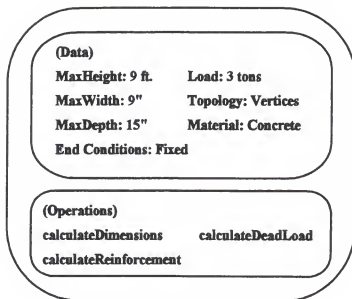


Figure 40. An example of a simple column object.

For example, a column is a physical architectural entity. A column can be modeled as a computational object (see Figure 40). The data of the column object comprises its topology, its dimensions, its loading conditions, its dimensional constraints and its material specification. The operations of the column include a method to size itself based on its loading conditions and constraints. The operations also include methods for the column to format itself in different structural systems. Column objects can interact with structural system objects to be sized according to loads on the structural system. Column objects can also interact with beam objects to define structural systems like a simple frame structure. Column objects can also maintain an internal mechanism that administers constraints when the column object is executing methods to size itself.

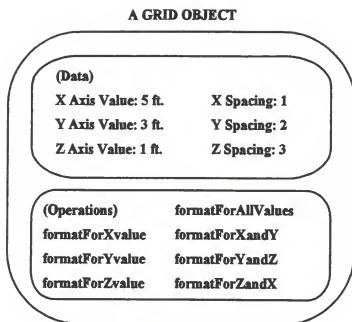


Figure 41. An example of a simple grid object.

The grid is a conceptual architectural entity. A grid can be modeled as a computational object (see Figure 41). The data of a Cartesian grid are the grid values along the three coordinate axes. The operations of the grid object include formatting other objects with the grid values along the three coordinate axes. Grids in two dimensions and grids with alternating grid values, as in a tartan grid, can also be modeled in this way. Different grid patterns with different rhythms, i.e., different sets of alternating values for the grid cells can also be modeled. Grids are essentially place holders for other architectural objects. Grid objects can also interact with other grid objects to form complex field objects. The interaction of grid objects can actually produce or instantiate field objects. Grid objects can also produce representations of the grids they represent.

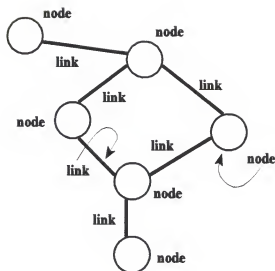


Figure 42. Graph representation of a circulatory system.

A circulatory system can be computationally modeled using graph theory (see Figure 42). The data of the graph include its nodes and its edges. The node can represent a space and the edges can represent links between spaces. Methods that operate on the graph's data include finding the centrality of a node (the Konig number), finding the shape index of the graph, finding the beta index of the graph and optimizing the graph for minimum circulation distances. Duals of graphs (Broadbent, 1973) or Teague networks (Mitchell, 1977) can be used to derive spatial enclosure patterns that reflect circulation patterns represented by the graphs (see Figure 43). Ordering system can also be computationally represented using graph theory. The data in an ordering system consist basically of connectivity information. The data represent adjacencies of spaces.

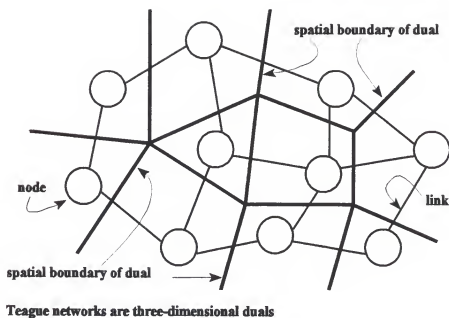


Figure 43. Dual representation of a graph.

Physical structures can be modeled using the methods of solid modeling, and spatial enclosures can be modeled as closed volumes using the methods of void modeling (Yessios, 1987).

Interaction of Architectural Computational Objects

Architectural designs in an ideal object-oriented computer-based design system are synthesized by the interaction of computational objects. This interaction can be achieved in many ways. One of them is through the use of a visual language (Shu, 1988). According to Shu, the use of visual languages is a new paradigm for expressing system computations that offers the possibility of directly manipulating computational objects.

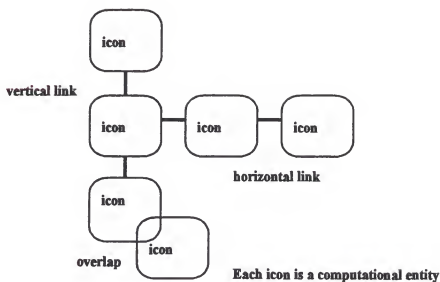


Figure 44. A visual program.

A visual program or a visual sentence is written in a visual language by a spatial arrangement of icons that represent computational objects. The spatial arrangements can be literal or metaphoric representations of the systems to be designed. Traditional programs are written as a sequence of instructions. The constructional operation in putting these programs together is concatenation. In a visual language, because a two-dimensional space is used, the constructional operations involve horizontal linking, vertical linking and spatial overlaps (see Figure 28). The visual interaction is used to develop the syntax of the program. Because the syntax used is visual, problems that can be solved by visual thinking or visual operations can be easily modeled in a visual language. When architects work in section or in plan, they are, in effect, solving problems in a two-dimensional visual language.

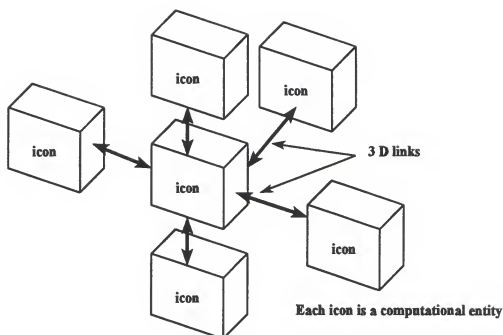


Figure 45. A visual program in three dimensions.

One can expand this concept and imagine a three-dimensional visual language with additional spatial construction operations (see Figure 45). Architects, in essence, work with such visual languages by spatially arranging building materials and products. Using visual programming or a visual language as the basis for a computer-based design system in architecture utilizes the mapping of analogous processes. Just as there is a syntax for programming, so there is a syntax for building. Using visual icons of architectural objects (both conceptual and physical) to synthesize architectural designs is a natural way to explicate and explore the syntax of building. The same rigor used in writing computer programs can be applied to the design decisions made by architects. Architects can program a building, as they often do in another sense.

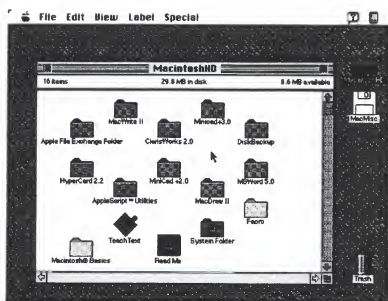


Figure 46. Printout of the screen of a Macintosh computer showing the desktop metaphor.

The desktop metaphor used in the AppleTM/MacintoshTM interface can provide a visual basis for structuring this interaction (see Figure 46). Icons representing architectural objects can be presented on the screen. The designer can then click on one of the icons, drag it over to another icon and click on it to set an interaction in motion. For example, an icon can represent the spatial enclosure of an auditorium. There can be another icon that represents a grid object. When the designer clicks on the grid object and then drags it and clicks on the auditorium object, the spatial enclosure is formatted according to the grid values in the grid object. Further, a spatial enclosure object can interact with a structural system object to define the dimensions of the structural system. The structural system object can then compute the dimensions of its individual members. Each of these computational objects should have methods defined for interaction with all other relevant computational objects. This will define a language of interaction for each computational object. Synthesizing an architectural design

by the interaction of computational objects uses a connectionist model and generates designs by using what Bakhtin (1981) calls dialogic mediation.

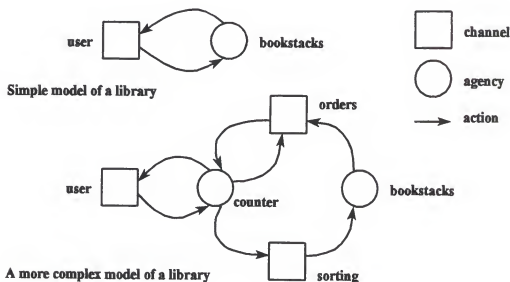


Figure 47. Models of a library using channel-agency nets (after Reisig, 1992).

Another model for structuring the interaction of computational objects is the use of Petri nets. Reisig (1992) discusses Petri nets in detail in his book on the subject. Petri nets were introduced in the 1970s as channel-agency nets (see Figure 47). The channels were the passive components, and the agencies were the active components. The state and behavior of computational objects can be mapped onto channels and agencies, respectively. Petri nets were introduced to overcome the drawbacks of flow charts that were being used to model computational tasks. Petri nets are used in the initial stages of system design to model hardware, communication protocols, parallel programs and distributed databases, all of which

involve complex interactions. Petri nets are effective not only to model computer systems but any organizational system. Architectural design can be conceived of as organization, hence it can be represented by Petri nets. Petri net modeling enables the checking of the formal correctness of the system being modeled. It also enables the derivation of precise mapping rules that can be used to generate algorithms from the formal specification of the system. Petri nets are strict bipartite graphs with the underlying mathematical model and semantics. The use of Petri nets ensures that a mathematical model can be established for the system being modeled. This makes the system amenable to computation. There are different kinds of Petri nets. These include condition-event nets, place-transition nets, individual-token nets and channel-agency nets. These nets are used to model different aspects of systems. It is possible to switch the model of a system from a channel-agency net to the other kinds of nets. These different kinds of Petri nets and their relationships are described in detail by Reisig (1992). The study of Petri nets is becoming increasingly important and there are annual international conferences on the applications and theory of Petri nets. As such Petri nets are a promising model with which to structure the synthesizing interaction of computational objects for architectural design.

Benefits of Object-oriented Design Systems in Architecture

There are many benefits in using the object-oriented paradigm for the development of computer-based design systems in architecture. The implemented design systems for the preliminary spatial design of proscenium-type auditoria reflect only some of the benefits. A

larger range of benefits can be realized that can be grouped into categories. These additional benefits are to be realized in future implementations of similar design systems.

The Object-oriented Perspective

An object-oriented approach forces the architect to think in terms of architectural objects and their characteristics. The object-oriented analysis of architectural designs provides fresh insight as to how architects manipulate architectural objects. The definition of the state and behavior of architectural objects explicates design knowledge. Architects are forced to rationalize their decision making process when they synthesize architectural objects. The multi-dimensional aspects of architectural objects can be modeled in an object-oriented approach that gives the designer a holistic perspective. These factors can improve the quality of computer-based design systems in architecture. The new knowledge aids future design decision making. The object-oriented analysis of conceptual objects in architecture has great promise for architectural research. In the implemented systems, the spatial form of the auditorium was abstracted as a computational object.

Abstraction

The object-oriented approach allows the architect to model architectural objects as true abstractions. A column can be modeled as an object that supports. A beam object can be modeled as an object that transfers vertical loads horizontally. This allows the semantic manipulation of those objects. The interface of the column object can prescribe how it links to a beam object. Both objects can have internal representations of their spatial location and

dimensions that can be calculated based on the loads applied on them. Intelligent architectural objects can be developed that can compute their own shape and form. The class system in object-oriented computing can be used to create a hierarchy of beam objects or column objects that vary in form and function. This allows generalization and specialization in the abstraction of architectural objects.

Conventional systems force architects to model architectural elements as combinations and transformations of primitive solid geometric entities such as cubes, spheres, pyramids, wedges or as planar surfaces. These entities are manipulated as data structures consisting of a collection of vertices and edges that define them. They cannot be manipulated semantically, i.e., as beams or columns. The building blocks in conventional systems are data structures that represent geometric entities. The object-oriented approach can help create architectural objects that are abstractions at a higher level than geometric entities and are more naturally manipulated by architects. Such abstractions can also allow decision making based on semantics.

Fuzzy Definitions

Fuzzy definitions of architectural elements in design decision making allows the sharing of responsibility between different participants of the design process. During the design process, architects collaborate with many specialists who help to design various parts of the project. For example, structural engineers help design the structural systems and mechanical engineers help design the air-conditioning systems. In the object-oriented approach, the architect can create objects that represent the parts to be designed by others,

develop the interface to those objects, specify constraints and leave it to the specialists to develop the object in detail. The architect working on the overall design is not bothered by the details of subordinate architectural objects. This facility allows the smoother coordination of the design process when there are a team of designers. A similar need for fuzzy definitions was expressed by Eastman (1987).

Context Sensitive Design Decision Making

Architectural objects are polymorphic. They have different functions in different contexts. A wall, which is an architectural object, can be an element of enclosure, a thermal barrier, an acoustical surface, a structural component and a visual object that has aesthetic proportions among other things. Depending on the context, an architect is interested in making decisions based on considering the wall in any one of those forms. By mapping the state and behavior of an architectural object into context related groups (Figure 4), a context sensitive interface can be developed for those objects. This kind of context-based abstraction is available only in the object-oriented approach. Context-based abstraction also helps the analysis of ensembles of architectural objects in a particular context mode. For example, all architectural objects can be analyzed in the structural mode to do structural analysis.

Multiple Representations

In the object-oriented approach, frameworks of objects such as the model-view-controller of Smalltalk™ can be created to provide a system of multiple representations for a design. When an architect makes a design decision, he should be aware of its ramifications

in multiple contexts. If he moves a wall, he should be aware of the structural conditions that he has changed, the change in the acoustical properties of the room, the change in the daylight levels in the room, the change in the aesthetic proportions of related walls, the change in the degree of enclosure etc. An object-oriented computer-based design system can provide the multiple representations that represent all those changes based on context specific information in the wall object. The different representations are linked to one object, therefore, when that object is transformed, the different representations are revised. This feature helps create a dynamic design medium that facilitates integrated design decision making. This system also helps the architect to coordinate all his representations into a self-consistent whole. Conventional computer-based design systems do not support such context-based multiple representations.

The Use of Precedent

There is a wealth of exemplary architectural works existing in the world. Architectural objects can be defined based on those exemplary works. Subsequently created architectural objects can inherit the state and behavior of those exemplar objects and modify them for special applications. The concepts of class hierarchies and inheritance in the object-oriented approach can support the use of precedent in architectural design. The use of precedent is popular in the architectural profession. In a conventional system, representations of architectural elements that are created are reusable only in the same form, e.g., symbol libraries.

Integrated Design and Analysis

Computer-based design systems based on problem solving and constraint-based decision making involve modules that are used to represent candidate solutions and allow their transformation, and modules that test those solutions to determine if they are satisfactory. Conventional computer-aided drawing or drafting systems provide only the representational medium. The analysis and testing of those representations involves the use of additional software. Separate software is also needed to monitor the search process and administer the constraints. Since the representations contain only limited descriptive data, all other required information is stored in a relational data base. The coordination between the different modules and the relational data base is a cumbersome process. The object-oriented approach with encapsulated state and behavior (data and operations) can solve this problem.

The main tasks of the designers and implementors of object-oriented computer-based design systems are the definition of architectural objects and the structuring of their interaction. The modular approach of using objects allows the designers and implementors of object-oriented computer-based design systems to concentrate on each object and its characteristics. The behavior of architectural objects based on their interaction with other objects in different contexts has to be modeled. This is a very complex task initially but has its rewards later. The structuring of the interaction of architectural objects involves the formulation of design strategies. Established design strategies can be organized into frameworks which can be reused. These strategies will clearly reflect how architects synthesize designs. Formulation of the design strategies will explicate the decision making

part of the architectural design process. This will lead to the development of first-order design systems in architecture. The seamless integration of the analysis, design and implementation stages (Korson & McGregor, 1990) in the object-oriented approach helps the rapid development of prototype design systems. Inheritance and polymorphism help the extension of existing design systems. This was verified in the implementation of the design system for rectangular proscenium-type auditoria.

Future Directions of Research

This dissertation sets up the grounds for significant research in two areas. The first area is the extension of the research in acoustic sculpting. The second area is the research on the object-oriented modeling of architectural design. Significant new work has been done in areas related directly to acoustic sculpting (Chiang, 1994). The object-oriented modeling of architectural design is still in its infancy. The following two sections present future directions of research in these two areas.

Acoustic Sculpting

The initial implementation of acoustic sculpting used a single sound source and a single receiver to generate the spatial form of the auditorium based on the acoustical parameters measured at the receiver location. This model has to be expanded to include multiple sound sources and multiple receiver locations. Methods have to be developed to optimize the spatial forms generated by the different sets of acoustical parameters that will

be used in such a model. A simple method to optimize the spatial forms is by using Boolean operators like union, intersection and difference.

In the initial implementation of acoustic sculpting, parameters were not used from all five subjective perception categories that were defined. In future implementations of acoustic sculpting, parameters from all five subjective categories must be used. The categories are reverberance, clarity, loudness, balance and spatiality or envelopment. Parameters associated with these subjective categories were treated with equal footing in the initial implementation of acoustic sculpting. Research has to be done to see how these subjective categories combine to result in the perception of overall acoustical quality. This will modify the generative system for the spatial design of the auditorium. A ray tracing model for the simulation of sound propagation in the computer model of the auditorium must be developed. This model should be used to generate energy impulse response graphs that can be convolved with a sound signal to predict what the space represented by the computer model will sound like if it is built.

Object-oriented Modeling of Architectural Design

A computational object representing a proscenium-type auditorium was implemented in the dissertation. Work is to be done to create several other architectural objects, some of which were described in Chapter 2. The complete definition of these objects is to be attempted along with their language of interaction. An environment for creating visual languages must also be developed to provide the medium for structuring the synthesizing interaction of the architectural objects. Several other models of interaction must be explored,

e.g., Petri net design. Research has to be done to establish an object-oriented database to manage the synthesized architectural object.

APPENDIX A ACOUSTICAL DATA SOURCE

Part of the acoustical data used in the calibration of the auditorium model are based on the data set reported in doctoral dissertation of Chiang (1994). The data set is presented as Table B-1, and the parameters used in the data set are described in Table 4-1 in Chiang's dissertation. The list of spaces in which the acoustical measurements were made is described in Table 2-2 in Chiang's dissertation. The procedure used to collect the data is described in the pamphlet on the A.R.I.A.S. (Acoustical Research Instrumentation for Architectural Spaces) system published by Doddington, Schwab, Siebein, Cervone and Chiang who were part of the research team on architectural acoustics at the University of Florida during the time that this data were collected. Using this data set the following relationships were established between the wall splay angle of the auditorium and the IACC (Inter Aural Cross Correlation) and Treble Ratio parameters using simple linear regression models:

$$1) \text{ wallSplayAngle} = ((\text{IACC} - 0.284) / (0.005 * \text{auditoriumDepth})) \arctan \text{ abs}$$

(This relationship was established with an R^2 of 0.3312 and a $\text{Prob} > |T|$ of 0.0125)

$$2) \text{ wallSplayAngle} = ((0.949 - \text{TrebleRatio}) / (0.002 * \text{auditoriumDepth})) \arctan \text{ abs}$$

(This relationship was established with an R^2 of 0.2540 and a $\text{Prob} > |T|$ of 0.0330)

Both the parameters were correlated with the width increase caused by the wall splay angle of the auditorium. This width increase was computed from the relationship:

$\text{width increase} = \text{auditoriumDepth} * \tan(\text{wallSplayAngle})$

The IACC and Treble Ratio parameters are described in Table 3-1 of Chiang's dissertation.

APPENDIX B

COMPUTER CODE FOR THE DESIGN SYSTEMS

The following is the complete computer code for the design systems written in the Smalltalk programming language in the VisualWorks™ environment:

```
View subclass: #AuditoriumFrameView
    instanceVariableNames: "
    classVariableNames: "
    poolDictionaries: "
    category: 'Auditorium'
```

AuditoriumFrameView methodsFor: 'displaying'

```
displayOn: aGraphicsContext
    "displays an shaded plane image of the auditorium on a GraphicsContext"

    | pos |
    pos := self bounds center.
    self model planes do: [:each |
        aGraphicsContext
            displayPolyline: (each points collect: [:i | i extractPointWith: self model
viewingPlaneDistance value])
            at: pos].
    ^self
```

AuditoriumFrameView methodsFor: 'updating'

```
update: aParameter
    "updates the auditorium frame view"

    self invalidate
```

```

View subclass: #AuditoriumPlaneView
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Auditorium'

```

AuditoriumPlaneView methodsFor: 'displaying'

```

displayOn: aGraphicsContext
  "displays an shaded plane image of the auditorium on a GraphicsContext"

  | pos |
  pos := self bounds center.
  self model setColoredPlanes do: [:each |
    aGraphicsContext
      paint: each color;
      displayPolygon: (each points collect: [:i | i extractPointWith: self model
viewingPlaneDistance value])
      at: pos].
  ^self

```

AuditoriumPlaneView methodsFor: 'updating'

```

update: aParameter
  "updates the auditorium plane view"

  self invalidate

```

ApplicationModel subclass: #Auditorium

instanceVariableNames: 'eyepoint lightpoint eyepointDistance eyepointLatitude eyepointLongitude lightpointDistance lightpointLatitude lightpointLongitude viewingPlaneDistance stageDepth stageWidth stageHeight prosceniumWidth prosceniumHeight apronDepth auditoriumDepthFromVisualClarity seatingSlope auditoriumCapacity areaPerSeat performanceMode timeDelay1 timeDelay2 timeDelay3 timeDelay4 reverberationTime loudnessLossAllowable iacc trebleRatio planes planeView frameView dataReport'

classVariableNames: "

poolDictionaries: "

category: 'Auditorium'

Auditorium methodsFor: 'compiling'

compileDataReport

"compiles the auditorium data report"

| aStream |

aStream := ReadWriteStream on: ".

aStream nextPutAll: 'Scroll through this screen for auditorium data:'; cr;

nextPutAll: 'Auditorium Volume (cft): ';

nextPutAll: self auditoriumVolume printString; cr;

nextPutAll: 'Approximate Wall and Roof Surface Area (sft): ';

nextPutAll: self approximateWallAndRoofSurfaceArea printString; cr;

nextPutAll: 'Room Constant: ';

nextPutAll: self roomConstant printString; cr;

nextPutAll: 'Average Absorption Coefficient: ';

nextPutAll: self averageAbsorptionCoefficient printString; cr;

nextPutAll: 'Average Wall Absorption Coefficient: ';

nextPutAll: self averageWallAbsorptionCoefficient printString; cr;

nextPutAll: 'Auditorium Depth (ft): ';

nextPutAll: self auditoriumDepth printString; cr;

nextPutAll: 'Average Auditorium Height (ft): ';

nextPutAll: self averageAuditoriumHeight printString; cr;

nextPutAll: 'Average Auditorium Width (ft): ';

nextPutAll: self averageAuditoriumWidth printString; cr;

nextPutAll: 'Front Row Distance (ft): ';

nextPutAll: self frontRowDistance printString; cr;

nextPutAll: 'Seating Area (sft): ';

nextPutAll: self seatingArea printString; cr;

nextPutAll: 'Balcony Seating Area (sft): ';

```

nextPutAll: self balconyArea printString; cr;
nextPutAll: 'Balcony Shortfall: ';
nextPutAll: self balconyShortfall printString,' % ', 'of seating area'; cr;
nextPutAll: 'Balcony Clearance Height (ft): ';
nextPutAll: self balconyClearanceHeight printString; cr;
nextPutAll: 'Balcony Depth (ft): ';
nextPutAll: self balconyDepth printString; cr;
nextPutAll: 'Wall Splay Angle (deg): ';
nextPutAll: self wallSplayAngle radiansToDegrees printString; cr;
nextPutAll: 'Seating Slope Angle (deg): ';
nextPutAll: self seatingSlopeAngle radiansToDegrees printString; cr;
nextPutAll: 'Seating Height (ft): ';
nextPutAll: self seatingHeight printString; cr;
nextPutAll: 'Proscenium Width (ft): ';
nextPutAll: self prosceniumWidth printString; cr;
nextPutAll: 'Proscenium Height (ft): ';
nextPutAll: self prosceniumHeight printString; cr;
nextPutAll: 'Stage Depth (ft): ';
nextPutAll: self stageDepth printString; cr;
nextPutAll: 'Stage Height (ft): ';
nextPutAll: self stageHeight printString; cr;
nextPutAll: 'Stage Width (ft): ';
nextPutAll: self stageWidth printString; cr;
nextPutAll: 'First Roof Segment Height (ft): ';
nextPutAll: self roofSegment1Height printString; cr;
nextPutAll: 'Second Roof Segment Height (ft): ';
nextPutAll: self roofSegment2Height printString; cr;
nextPutAll: 'Third Roof Segment Height (ft): ';
nextPutAll: self roofSegment3Height printString; cr;
nextPutAll: 'Fourth Roof Segment Height (ft): ';
nextPutAll: self roofSegment4Height printString; cr.
^aStream contents

```

Auditorium methodsFor: 'setting'

```

prosceniumHeight: aHeight
    "sets the proscenium height of the auditorium to be aHeight"

    prosceniumHeight := aHeight

prosceniumWidth: aWidth
    "sets the proscenium width of the auditorium to be aWidth"

```

prosceniumWidth := aWidth

setDataReportAndPlanes

"sets the data report and the planes of the auditorium"

self dataReport value: self compileDataReport.

self setPlanes.

setDrama

"sets the performance mode to drama"

self performanceMode value: 'drama'

setEyepoint

"sets the eyepoint of the auditorium"

eyepoint := (EyePoint new) distance: self eyepointDistance value latitude: self
eyepointLatitude value longitude: self eyepointLongitude value.

self setPlanes

setLightpoint

"sets the lightpoint of the auditorium"

lightpoint := (LightPoint new) distance: self lightpointDistance value latitude: self
lightpointLatitude value longitude: self lightpointLongitude value.

self setPlanes

setMusical

"sets the performance mode to musical"

self performanceMode value: 'musical'

setOpera

"sets the performance mode to opera"

self performanceMode value: 'opera'

setPlanes

"sets the planes that define the shape of the auditorium and sets the instance variable
planes"

planes := OrderedCollection new.

```

    planes add: self plane1; add: self plane2; add: self plane3; add: self plane4; add: self
plane5; add: self plane6; add: self plane7; add: self plane8;
    add: self plane9; add: self plane10; add: self plane11; add: self plane12; add: self
plane13; add: self plane14; add: self plane15; add: self plane16;
    add: self plane17; add: self plane18; add: self plane19; add: self plane20; add: self
plane21; add: self plane22; add: self plane23; add: self plane24;
    add: self plane25; add: self plane26; add: self plane27; add: self plane28; add: self
plane29; add: self plane30; add: self plane31; add: self plane32.
    self changed

```

setProsceniumDimensions

"sets the proscenium dimensions of the auditorium based on the performance mode of the auditorium"

```

    self performanceMode value = 'theater'
        ifTrue: [self prosceniumWidth: 28; prosceniumHeight: 20]
        ifFalse: [self performanceMode value = 'drama'
            ifTrue: [self prosceniumWidth: 35; prosceniumHeight: 17.5]
            ifFalse: [self performanceMode value = 'musical'
                ifTrue: [self prosceniumWidth: 45; prosceniumHeight: 25]
                ifFalse: [self performanceMode value = 'opera'
                    ifTrue: [self prosceniumWidth: 70; prosceniumHeight: 25]
                    ifFalse: [self prosceniumWidth: 75; prosceniumHeight: 30]]]]

```

setStageDimensions

"sets the stage dimensions of the auditorium based on standards"

```

    stageDepth := self prosceniumWidth*1.25.
    stageHeight := (self prosceniumHeight*2.75) + 9.
    stageWidth := self prosceniumWidth*2.5

```

setStageDimensionsAndPlanes

"sets the stage and proscenium dimensions, and the planes of the auditorium"

```

    self setProsceniumDimensions; setStageDimensions; setPlanes.

```

setStageDimensionsReportAndPlanes

"sets the stage and proscenium dimensions, data report and the planes of the auditorium"

```

    self setProsceniumDimensions; setStageDimensions; setDataReportAndPlanes.

```

setSymphony

"sets the performance mode to symphony"

self performanceMode value: 'symphony'

setTheater

"sets the performance mode to theater"

self performanceMode value: 'theater'

setTimeDelay1

"sets the time delay of the first reflection based on anInterval"

| ellipseMajorAxis ellipseMinorAxis eccentricity |
 ellipseMajorAxis := (self auditoriumDepth + (self stageDepth*0.5))*0.5.
 ellipseMinorAxis := (ellipseMajorAxis*self prosceniumHeight) sqrt.
 eccentricity := (1 - (ellipseMinorAxis squared/ellipseMajorAxis squared)) sqrt.
 self timeDelay1 value: ((self timeDelay1 value) max: (((ellipseMajorAxis*2) -
 (ellipseMajorAxis*2*eccentricity))/1130))

setTimeDelay2

"sets the time delay of the second reflection based on anInterval"

self timeDelay2 value: ((self timeDelay2 value) max: (self timeDelay1 value))

setTimeDelay3

"sets the time delay of the third reflection based on anInterval"

self timeDelay3 value: ((self timeDelay3 value) max: (self timeDelay2 value))

setTimeDelay4

"sets the time delay of the fourth reflection based on anInterval"

self timeDelay4 value: ((self timeDelay4 value) max: (self timeDelay3 value))

stageDepth: aDepth

"sets the stage depth of the auditorium to be aDepth"

stageDepth := aDepth

stageHeight: aHeight

"sets the stage height of the auditorium to be aHeight"

stageHeight := aHeight

stageWidth: aWidth

"sets the stage width of the auditorium to be aWidth"

stageWidth := aWidth

Auditorium methodsFor: 'accessing'

approximateWallAndRoofSurfaceArea

"returns the approximate wall and roof surface area of the auditorium assuming flat roof segments and neglecting the strip area around the proscenium"

```
| p q r s t u surfaceArea |
p := (self prosceniumWidth + 12)*(self wallSplayAngle cos*self auditoriumDepth).
q := (self wallSplayAngle cos + self wallSplayAngle sin)*self auditoriumDepth.
r := ((self prosceniumWidth*0.5) + 6 + (self wallSplayAngle sin*self auditoriumDepth))*(self auditoriumDepth - (self wallSplayAngle cos*self auditoriumDepth)).
s := (self auditoriumDepth - (self wallSplayAngle cos*self auditoriumDepth))/self wallSplayAngle sin.
t := (self balconyClearanceHeight + 9)*s*2.
u := self averageAuditoriumHeight*self auditoriumDepth*2.
surfaceArea := (p + q + r + t + u).
^surfaceArea
```

auditoriumDepth

"returns the allowable depth of the auditorium optimizing for constraints"

^(self auditoriumDepthFromLoudness) min: (self auditoriumDepthFromVisualClarity value)

auditoriumDepthFromLoudness

"returns the depth of the auditorium based on loudness loss allowable"

^self loudnessLossAllowable value/0.049

auditoriumVolume

"returns the volume of the auditorium subtracting the balcony volume"

```
| s balconyVolume auditoriumVolume |
```

self wallSplayAngle = 0

ifTrue: [

s := (self prosceniumWidth*0.5) + 6]

ifFalse: [

```
s := (self auditoriumDepth - (self wallSplayAngle cos*self auditoriumDepth))/self
wallSplayAngle sin].
```

```
balconyVolume := s*self balconyDepth*self balconySeatingHeight.
```

```
auditoriumVolume := self averageAuditoriumHeight *self floorSeatingArea.
```

```
^auditoriumVolume - balconyVolume
```

averageAbsorptionCoefficient

"returns the average absorption coefficient for materials to be used on all wall and roof surfaces in the auditorium"

```
^(self roomConstant - (self floorSeatingArea*0.3*0.03))/ (self
approximateWallAndRoofSurfaceArea)
```

averageAuditoriumHeight

"returns the average height of the auditorium"

```
| r1 r2 r3 r4 h1 h2 h3 h4 h5 h6 averageHeight |
r1 := ((self roofSegment1Depth - self frontRowDistance) max: 0)*self
seatingSlopeAngle tan.
r2 := ((self roofSegment2Depth - self frontRowDistance) max: 0)*self
seatingSlopeAngle tan.
r3 := ((self roofSegment3Depth - self frontRowDistance) max: 0)*self
seatingSlopeAngle tan.
r4 := ((self roofSegment4Depth - self frontRowDistance) max: 0)*self
seatingSlopeAngle tan.
h1 := self prosceniumHeight + 12.5.
h2 := self roofSegment1Height + 9 - r1.
h3 := self roofSegment2Height + 9 - r2.
h4 := self roofSegment3Height + 9 - r3.
h5 := self roofSegment4Height + 9 - r4.
h6 := self balconyClearanceHeight + self balconySeatingHeight + 9.
averageHeight := (h1 + h2 + h3 + h4 + h5 + h6)*0.167.
^averageHeight
```

averageAuditoriumWidth

"returns the average width of the auditorium"

```
| w1 w2 averageWidth |
w1 := self prosceniumWidth + 12.
w2 := (self auditoriumDepth*self wallSplayAngle sin*2) + w1.
averageWidth := (w1 + w2)*0.5.
^averageWidth
```

averageWallAbsorptionCoefficient

"returns the average absorption coefficient for materials to be used on just the wall surfaces in the auditorium"

```
| s t u wallSurfaceArea |
s := (self auditoriumDepth - (self wallSplayAngle cos*self auditoriumDepth))/self
wallSplayAngle sin.
t := (self balconyClearanceHeight + 9)*s*2.
u := self averageAuditoriumHeight*self auditoriumDepth*2.
wallSurfaceArea := t + u.
^(self roomConstant - (self floorSeatingArea*0.3*0.03))/(wallSurfaceArea)
```

balconyArea

"returns the balcony area of the auditorium adjusted for constraints"

```
self wallSplayAngleBasedOnSeatingArea > 30
ifTrue: [^(1 - (30.0/self wallSplayAngleBasedOnSeatingArea))*self seatingArea) min:
(self seatingArea*0.3)]
ifFalse: [^0.0]
```

balconyClearanceHeight

"returns the balcony clearance height of the auditorium"

```
| cantileverClearanceAngle cantileverClearance |
cantileverClearanceAngle := ((self prosceniumHeight + 3.5 - self seatingHeight -
3.75)/self auditoriumDepth) arcTan.
cantileverClearanceAngle < 0
ifTrue: [cantileverClearance := 0]
ifFalse: [cantileverClearance := cantileverClearanceAngle tan*self balconyDepth].
self balconyArea = 0
ifFalse: [^(cantileverClearance + 4.75) max: ((self balconyDepth/1.5) - (self
seatingSlopeAngle tan*self balconyDepth))) max: 7.0]
ifTrue: [^0.0]
```

balconyDepth

"returns the balcony depth of the auditorium adjusted for constraints"

```
| seatingDepthFactor |
seatingDepthFactor := ((4*self auditoriumDepth squared) - (self balconyArea*2))
sqrt.
self balconyArea = 0
ifFalse: [^(self auditoriumDepth*2) - seatingDepthFactor) min: (self
auditoriumDepth*0.33)]
```

ifTrue: [^0.0]

balconySeatingHeight

"returns the balcony seating height of the auditorium"

self wallSplayAngleBasedOnSeatingArea > 30

ifTrue: [^self balconyDepth*0.577]

ifFalse: [^0.0]

balconyShortfall

"returns the percentage of the seating area shortfall due to the balcony area constraint"

| seatingDepthFactor actualBalconyArea |

seatingDepthFactor := ((4*self auditoriumDepth squared) - (self balconyArea*2))

sqrt.

actualBalconyArea := 0.5*((4*self auditoriumDepth squared) - (seatingDepthFactor squared)).

^((((1 - (30.0/self wallSplayAngleBasedOnSeatingArea))*self seatingArea) - (actualBalconyArea))/(self seatingArea))*100 max: 0.0

eyepoint

"returns the eyepoint of the auditorium"

^eyepoint

floorSeatingArea

"returns the floor seating area of the auditorium"

| p q r floorArea |

p := (self prosceniumWidth + 12)*(self wallSplayAngle cos*self auditoriumDepth).

q := (self wallSplayAngle cos + self wallSplayAngle sin)*self auditoriumDepth.

r := ((self prosceniumWidth*0.5) + 6 + (self wallSplayAngle sin*self auditoriumDepth))*(self auditoriumDepth - (self wallSplayAngle cos*self auditoriumDepth)).

floorArea := p + q + r.

^floorArea

frameView

"returns a frame view of the auditorium"

^frameView

frontRowDistance

"calculates and returns the front row distance from the proscenium"

^self apronDepth value + (6*1.732)

lightpoint

"returns the lightpoint of the auditorium"

^lightpoint

plane1

"sets the first plane that defines the shape of the auditorium"

| m n o p x points |

x := 0.000001.

m := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self stageDepth) negated withZ: (x + 9) negated.

n := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth) negated withZ: (x + 9) negated.

o := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth) negated withZ: ((x + self stageHeight) - 9).

p := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self stageDepth) negated withZ: ((x + self stageHeight) - 9).

points := (OrderedCollection new).

points add: m; add: n; add: o; add: p; add: m.

^Plane withId: 1 withPoints: points

plane10

"sets the tenth plane that defines the shape of the auditorium"

| m n o p x points |

x := 0.000001.

m := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 9) negated.

n := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ: (x + 9) negated.

o := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self apronDepth value) withZ: (x + 9) negated.

p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self apronDepth value) withZ: (x + 9) negated.

points := (OrderedCollection new).

points add: m; add: n; add: o; add: p; add: m.

^Plane withId: 10 withPoints: points

plane11

"sets the eleventh plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 5.5)
negated.
n := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
o := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self
apronDepth value) withZ: (x + 9) negated.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 9)
negated.
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 11 withPoints: points

```

plane12

"sets the twelfth plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
(x + 5.5) negated.
n := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
o := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self
apronDepth value) withZ: (x + 9) negated.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
(x + 9) negated.
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 12 withPoints: points

```

plane13

"sets the thirteenth plane that defines the shape of the auditorium"

```

| m n o p q r s t u x points |
x := 0.000001.
m := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: (x + 9) negated.
n := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
(x + 9) negated.
o := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self
apronDepth value) withZ: (x + 9) negated.

```

p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self apronDepth value) withZ: (x + 9) negated.

q := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 9) negated.

r := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (x + 9) negated.

s := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self frontRowDistance) + 6) withY: (x + (self wallSplayAngle cos*self frontRowDistance)) withZ: (x + 9) negated.

t := PointVector withX: x withY: (self frontRowDistance) withZ: (x + 9) negated.

u := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self frontRowDistance) + 6) negated withY: (x + (self wallSplayAngle cos*self frontRowDistance)) withZ: (x + 9) negated.

points := (OrderedCollection new).

points add: m; add: n; add: o; add: p; add: q; add: r; add: s; add: t; add: u; add: m.

^Plane withId: 13 withPoints: points

plane14

"sets the fourteenth plane that defines the shape of the auditorium"

| m n o p x points |

x := 0.000001.

m := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self frontRowDistance) + 6) withY: (x + (self wallSplayAngle cos*self frontRowDistance)) withZ: (x + 9) negated.

n := PointVector withX: x withY: (x + self frontRowDistance) withZ: (x + 9) negated.

o := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (x + self seatingHeight - 9).

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (x + self seatingHeight - 9).

points := (OrderedCollection new).

points add: m; add: n; add: o; add: p; add: m.

^Plane withId: 14 withPoints: points

plane15

"sets the fifteenth plane that defines the shape of the auditorium"

| m n o p x points |

x := 0.000001.

m := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self frontRowDistance) + 6) negated withY: (x + (self wallSplayAngle cos*self frontRowDistance)) withZ: (x + 9) negated.

```

n := PointVector withX: x withY: (x + self frontRowDistance) withZ: (x + 9) negated.
o := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (x + self
seatingHeight - 9).
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (x + self seatingHeight - 9).
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 15 withPoints: points

```

plane16

"sets the sixteenth plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + x - 9).
n := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (self seatingHeight + x - 9).
o := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).
p := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + x - 9).
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 16 withPoints: points

```

plane17

"sets the seventeenth plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + x - 9).
n := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (self seatingHeight + x - 9).
o := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).

```

```

p:= PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + x - 9).
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 17 withPoints: points

```

plane18

"sets the eighteenth plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m:= PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + x - 9).
n:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).
o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*(self auditoriumDepth - self balconyDepth)) + 6) withY: (x + (self wallSplayAngle
cos*(self auditoriumDepth - self balconyDepth))) withZ: (self seatingHeight + self
balconyClearanceHeight + x - 9).
p:= PointVector withX: x withY: (x + self auditoriumDepth - self balconyDepth)
withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 18 withPoints: points

```

plane19

"sets the nineteenth plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m:= PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + x - 9).
n:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + x -
9).
o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*(self auditoriumDepth - self balconyDepth)) + 6) negated withY: (x + (self
wallSplayAngle cos*(self auditoriumDepth - self balconyDepth))) withZ: (self seatingHeight
+ self balconyClearanceHeight + x - 9).
p:= PointVector withX: x withY: (x + self auditoriumDepth - self balconyDepth)
withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).

```

```

points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 19 withPoints: points

```

plane2

"sets the second plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self
stageDepth) negated withZ: ((x + self stageHeight) - 9).
n := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: ((x +
self stageHeight) - 9).
o := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: (x + 9)
negated.
p := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self
stageDepth) negated withZ: (x + 9) negated.
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 2 withPoints: points

```

plane20

"sets the twentieth plane that defines the shape of the auditorium"

```

| m n o p x points |
x := 0.000001.
m:= PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x - 9).
n:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x -
9).
o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*(self auditoriumDepth - self balconyDepth)) + 6) withY: (x + (self wallSplayAngle
cos*(self auditoriumDepth - self balconyDepth))) withZ: (self seatingHeight + self
balconyClearanceHeight + x - 9).
p:= PointVector withX: x withY: (x + self auditoriumDepth - self balconyDepth)
withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 20 withPoints: points

```

plane21

"sets the twentyfirst plane that defines the shape of the auditorium"

```
| m n o p x points |
x := 0.000001.
m:= PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x - 9).
n:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self
balconySeatingHeight + x - 9).
o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*(self auditoriumDepth - self balconyDepth)) + 6) negated withY: (x + (self
wallSplayAngle cos*(self auditoriumDepth - self balconyDepth))) withZ: (self seatingHeight
+ self balconyClearanceHeight + x - 9).
p:= PointVector withX: x withY: (x + self auditoriumDepth - self balconyDepth)
withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 21 withPoints: points
```

plane22

"sets the twentysecond plane that defines the shape of the auditorium"

```
| m n o p x points |
x := 0.000001.
m:= PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x).
n:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight +
x).
o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x -
9).
p:= PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x - 9).
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 22 withPoints: points
```

plane23

"sets the twentythird plane that defines the shape of the auditorium"

```
| m n o p x points |
```

```
x := 0.000001.
```

```
m := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x).
```

```
n := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self
balconySeatingHeight + x).
```

```
o := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self
balconySeatingHeight + x - 9).
```

```
p := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x - 9).
```

```
points := (OrderedCollection new).
```

```
points add: m; add: n; add: o; add: p; add: m.
```

```
^Plane withId: 23 withPoints: points
```

plane24

"sets the twentyfourth plane that defines the shape of the auditorium"

```
| m n o p points x |
```

```
x := 0.000001.
```

```
m := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x).
```

```
n := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x).
```

```
o := PointVector withX: x withY: (x + self roofSegment4Depth) withZ: (self
roofSegment4Height + x).
```

```
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment4Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment4Depth)) withZ: (self roofSegment4Height + x).
```

```
points := (OrderedCollection new).
```

```
points add: m; add: n; add: o; add: p; add: m.
```

```
^Plane withId: 24 withPoints: points
```

plane25

"sets the twentyfifth plane that defines the shape of the auditorium"

```
| m n o p points x |
```

```
x := 0.000001.
```

```

m := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self
auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self
balconySeatingHeight + x).

```

```

n := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (self
seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x).

```

```

o := PointVector withX: x withY: (x + self roofSegment4Depth) withZ: (self
roofSegment4Height + x).

```

```

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment4Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment4Depth)) withZ: (self roofSegment4Height + x).

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

```

^Plane withId: 25 withPoints: points

```

plane26

"sets the twenty-sixth plane that defines the shape of the auditorium"

```

| m n o p x points |

```

```

x := 0.000001.

```

```

m:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment4Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment4Depth)) withZ: (self roofSegment4Height + x).

```

```

n := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment4Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment4Depth)) withZ: (self roofSegment4Height + x).

```

```

o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment3Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment3Depth)) withZ: (self roofSegment3Height + x).

```

```

p:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment3Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment3Depth)) withZ: (self roofSegment3Height + x).

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

```

^Plane withId: 26 withPoints: points

```

plane27

"sets the twenty-seventh plane that defines the shape of the auditorium"

```

| m n o p x points |

```

```

x := 0.000001.

```

```

m:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment3Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment3Depth)) withZ: (self roofSegment3Height + x).

```

```

n := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment3Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment3Depth)) withZ: (self roofSegment3Height + x).

```

```

o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment2Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment2Depth)) withZ: (self roofSegment2Height + x).

```

```

p:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment2Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment2Depth)) withZ: (self roofSegment2Height + x).

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

```

^Plane withId: 27 withPoints: points

```

plane28

"sets the twentyeighth plane that defines the shape of the auditorium"

```

| m n o p x points |

```

```

x := 0.000001.

```

```

m:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment2Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment2Depth)) withZ: (self roofSegment2Height + x).

```

```

n := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment2Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment2Depth)) withZ: (self roofSegment2Height + x).

```

```

o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment1Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment1Depth)) withZ: (self roofSegment1Height + x).

```

```

p:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment1Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment1Depth)) withZ: (self roofSegment1Height + x).

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

```

^Plane withId: 28 withPoints: points

```

plane29

"sets the twenty ninth plane that defines the shape of the auditorium"

```

| m n o p x points |

```

```

x := 0.000001.

```

```

m:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment1Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment1Depth)) withZ: (self roofSegment1Height + x).

```

```

n:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment1Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment1Depth)) withZ: (self roofSegment1Height + x).

```

```

o:= PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: (self prosceniumHeight + x - 2).

```

```

p:= PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (self
prosceniumHeight + x - 2).

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

```

^Plane withId: 29 withPoints: points

```

plane3

"sets the third plane that defines the shape of the auditorium"

```

| m n o p x points |

```

```

x := 0.000001.

```

```

m := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth)
negated withZ: ((x + self stageHeight) - 9).

```

```

n := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: ((x + self
stageHeight) - 9).

```

```

o := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: (x + 9) negated.

```

```

p := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth)
negated withZ: (x + 9) negated.

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

```

^Plane withId: 3 withPoints: points

```

plane30

"sets the thirtieth plane that defines the shape of the auditorium"

```

| m n o p q r s t u x points v w y |

```

```

x := 0.000001.

```

```

m:= PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (self
prosceniumHeight + x - 2).

```

```

n:= PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (x +
9) negated.

```

```

o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self frontRowDistance) + 6) withY: (x + (self wallSplayAngle cos*self
frontRowDistance)) withZ: (x + 9) negated.

```

p:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + x - 9).

q:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).

r:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*(self auditoriumDepth - self balconyDepth)) + 6) withY: (x + (self wallSplayAngle cos*(self auditoriumDepth - self balconyDepth))) withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).

s:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x - 9).

t:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x).

u:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment4Depth) + 6) withY: (x + (self wallSplayAngle cos*self roofSegment4Depth)) withZ: (self roofSegment4Height + x).

v:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment3Depth) + 6) withY: (x + (self wallSplayAngle cos*self roofSegment3Depth)) withZ: (self roofSegment3Height + x).

w:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment2Depth) + 6) withY: (x + (self wallSplayAngle cos*self roofSegment2Depth)) withZ: (self roofSegment2Height + x).

y:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment1Depth) + 6) withY: (x + (self wallSplayAngle cos*self roofSegment1Depth)) withZ: (self roofSegment1Height + x).

points := (OrderedCollection new).

points add: m; add: n; add: o; add: p; add: q; add: r; add: s; add: t; add: u; add: v; add: w; add: y; add: m.

^Plane withId: 30 withPoints: points

plane31

"sets the thirtyfirst plane that defines the shape of the auditorium"

| m n o p q r s t u x points v w y |

x := 0.000001.

m:= PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x withZ: (self prosceniumHeight + x - 2).

n:= PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x withZ: (x + 9) negated.

o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self frontRowDistance) + 6) negated withY: (x + (self wallSplayAngle cos*self frontRowDistance)) withZ: (x + 9) negated.

p:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + x - 9).

q:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).

r:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*(self auditoriumDepth - self balconyDepth)) + 6) negated withY: (x + (self wallSplayAngle cos*(self auditoriumDepth - self balconyDepth))) withZ: (self seatingHeight + self balconyClearanceHeight + x - 9).

s:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x - 9).

t:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight + x).

u:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment4Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self roofSegment4Depth)) withZ: (self roofSegment4Height + x).

v:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment3Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self roofSegment3Depth)) withZ: (self roofSegment3Height + x).

w:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment2Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self roofSegment2Depth)) withZ: (self roofSegment2Height + x).

y:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment1Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self roofSegment1Depth)) withZ: (self roofSegment1Height + x).

points := (OrderedCollection new).

points add: m; add: n; add: o; add: p; add: q; add: r; add: s; add: t; add: u; add: v; add: w; add: y; add: m.

^Plane withId: 31 withPoints: points

plane32

"sets the thirtysecond plane that defines the shape of the auditorium"

| m n o x points |

x := 0.000001.

```

m:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment4Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment4Depth)) withZ: (self roofSegment4Height + x).

```

```

n:= PointVector withX: x withY: (x +self roofSegment4Depth) withZ: (self
roofSegment4Height + x).

```

```

o:= PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment4Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment4Depth)) withZ: (self roofSegment4Height + x).

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: m.

```

```

^Plane withId: 32 withPoints: points

```

plane4

"sets the fourth plane that defines the shape of the auditorium"

```

| m n o p x points |

```

```

x := 0.000001.

```

```

m := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self
stageDepth) negated withZ: ((x + self stageHeight) - 9).

```

```

n := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: ((x +
self stageHeight) - 9).

```

```

o := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: ((x + self
stageHeight) - 9).

```

```

p := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth)
negated withZ: ((x + self stageHeight) - 9).

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

```

^Plane withId: 4 withPoints: points

```

plane5

"sets the fifth plane that defines the shape of the auditorium"

```

| m n o p x points |

```

```

x := 0.000001.

```

```

m := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self
stageDepth) negated withZ: (x + 9) negated.

```

```

n := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: (x + 9)
negated.

```

```

o := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: (x + 9) negated.

```

```

p := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth)
negated withZ: (x + 9) negated.

```

```

points := (OrderedCollection new).

```

```

points add: m; add: n; add: o; add: p; add: m.

```

^Plane withId: 5 withPoints: points

plane6

"sets the sixth plane that defines the shape of the auditorium"

```
| m n o p q r s t x points |
x := 0.000001.
m := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: ((x +
self stageHeight) - 9).
n := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: ((x + self
stageHeight) - 9).
o := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: (x + 9) negated.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (x +
9) negated.
q := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: ((x
+ self prosceniumHeight) - 9).
r := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: ((x + self prosceniumHeight) - 9).
s := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: (x + 9) negated.
t := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: (x + 9)
negated.
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: q; add: r; add: s; add: t; add: m.
^Plane withId: 6 withPoints: points
```

plane7

"sets the seventh plane that defines the shape of the auditorium"

```
| m n o p x points |
x := 0.000001.
m := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 5.5)
negated.
n := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
(x + 5.5) negated.
o := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
(x + 9) negated.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 9)
negated.
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 7 withPoints: points
```

plane8

"sets the eighth plane that defines the shape of the auditorium"

```
| m n o p q r s t x points |
x := 0.000001.
m := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self
stageDepth) negated withZ: (x + 5.5) negated.
n := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: (x +
5.5) negated.
o := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
(x + 5.5) negated.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
q := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
r := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 5.5)
negated.
s := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: (x + 5.5)
negated.
t := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth)
negated withZ: (x + 5.5) negated.
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: q; add: r; add: s; add: t; add: m.
^Plane withId: 8 withPoints: points
```

plane9

"sets the ninth plane that defines the shape of the auditorium"

```
| m n o p x points |
x := 0.000001.
m := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
n := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
o := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self
apronDepth value) withZ: (x + 9) negated.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self
apronDepth value) withZ: (x + 9) negated.
points := (OrderedCollection new).
points add: m; add: n; add: o; add: p; add: m.
^Plane withId: 9 withPoints: points
```

planes

"returns the collection of planes that make up the spatial form of the auditorium"

^planes

planeView

"returns a plane view of the auditorium"

^planeView

prosceniumHeight

"returns the height of the proscenium of the auditorium"

^prosceniumHeight

prosceniumWidth

"returns the width of the proscenium of the auditorium"

^prosceniumWidth

roofSegment1Depth

"returns the depth of the first roof segment"

^self auditoriumDepth*0.10

roofSegment1Height

"returns the height of the first roof segment"

| ellipseMajorAxis ellipseMinorAxis eccentricAngle seatingHeight |
 seatingHeight := (((self roofSegment1Depth - self frontRowDistance) max: 0)*self
 seatingSlopeAngle tan) - 9.

ellipseMajorAxis := (self auditoriumDepth + (self timeDelay1 value*1130))*0.5.

ellipseMinorAxis := ((ellipseMajorAxis squared - (self auditoriumDepth*0.5)
 squared)) sqrt.

eccentricAngle := (ellipseMinorAxis/ellipseMajorAxis) arcSin + 10 degreesToRadians.

^((ellipseMinorAxis*eccentricAngle sin) + seatingHeight) max: (self
 prosceniumHeight + 3.5)

roofSegment2Depth

"returns the depth of the second roof segment"

^self auditoriumDepth*0.2

roofSegment2Height

"returns the height of the second roof segment"

```
| ellipseMajorAxis ellipseMinorAxis eccentricAngle seatingHeight |
seatingHeight := (((self roofSegment2Depth - self frontRowDistance) max: 0)*self
seatingSlopeAngle tan) - 9.
ellipseMajorAxis := (self auditoriumDepth + (self timeDelay2 value*1130))*0.5.
ellipseMinorAxis := ((ellipseMajorAxis squared - (self auditoriumDepth*0.5)
squared)) sqrt.
eccentricAngle := (ellipseMinorAxis/ellipseMajorAxis) arcSin + 10 degreesToRadians.
^((ellipseMinorAxis*eccentricAngle sin) + seatingHeight) max: (self
prosceniumHeight + 3.5)
```

roofSegment3Depth

"returns the depth of the third roof segment"

```
^self auditoriumDepth*0.3
```

roofSegment3Height

"returns the height of the third roof segment"

```
| ellipseMajorAxis ellipseMinorAxis eccentricAngle seatingHeight |
seatingHeight := (((self roofSegment3Depth - self frontRowDistance) max: 0)*self
seatingSlopeAngle tan) - 9.
ellipseMajorAxis := (self auditoriumDepth + (self timeDelay3 value*1130))*0.5.
ellipseMinorAxis := ((ellipseMajorAxis squared - (self auditoriumDepth*0.5)
squared)) sqrt.
eccentricAngle := (ellipseMinorAxis/ellipseMajorAxis) arcSin + 10 degreesToRadians.
^((ellipseMinorAxis*eccentricAngle sin) + seatingHeight) max: (self
prosceniumHeight + 3.5)
```

roofSegment4Depth

"returns the depth of the fourth roof segment"

```
^self auditoriumDepth*0.4
```

roofSegment4Height

"returns the height of the fourth roof segment"

```
| ellipseMajorAxis ellipseMinorAxis eccentricAngle seatingHeight |
seatingHeight := (((self roofSegment4Depth - self frontRowDistance) max: 0)*self
seatingSlopeAngle tan) - 9.
ellipseMajorAxis := (self auditoriumDepth + (self timeDelay4 value*1130))*0.5.
```

ellipseMinorAxis := ((ellipseMajorAxis squared - (self auditoriumDepth*0.5) squared)) sqrt.

eccentricAngle := (ellipseMinorAxis/ellipseMajorAxis) arcSin + 10 degreesToRadians.

^(ellipseMinorAxis*eccentricAngle sin) + seatingHeight) max: (self prosceniumHeight + 3.5)

roomConstant

"returns the Room Constant of the walls and roof surfaces of the auditorium using a 50% occupancy rate, 70% seat area and taking into account absorption due to air"

^((0.049*self auditoriumVolume)/self reverberationTime value) - ((self auditoriumVolume/1000)*0.9) - (self floorSeatingArea*0.70*0.5*0.94) - (self floorSeatingArea*0.70*0.5*0.62)

seatingArea

"calculates and returns the seating area of the auditorium based on the capacity of the auditorium and the area per seat"

^self auditoriumCapacity value*self areaPerSeat value

seatingHeight

"returns the maximum height of the seating area of the auditorium from the base level"

^(self auditoriumDepth - self frontRowDistance)*(self seatingSlopeAngle tan)

seatingSlopeAngle

"calculates and returns the slope angle (in radians) of the seating area of the auditorium adjusted for constraints"

^(((5.5/self frontRowDistance) arcTan)*((self auditoriumDepth/self frontRowDistance ln)) min: (self seatingSlope value) degreesToRadians

stageDepth

"returns the depth of the stage of the auditorium"

^stageDepth

stageHeight

"returns the height of the stage of the auditorium"

^stageHeight

stageWidth

"returns the width of the stage of the auditorium"

^stageWidth

transMatrix

"returns the translation matrix based on the eyepoint of the viewer of the auditorium"

^TransMatrix viewing: self eyepoint

v1

"returns the first vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self stageDepth) negated withZ: (x + 9) negated.

^self computeScreenCoordinate: p

v10

"returns the tenth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self apronDepth value) withZ: (x + 9) negated.

^self computeScreenCoordinate: p

v11

"returns the eleventh vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self apronDepth value) withZ: (x + 9) negated.

^self computeScreenCoordinate: p

v12

"returns the twelfth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 9) negated.

^self computeScreenCoordinate: p

v13

"returns the thirteenth vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
(x + 5.5) negated.
^self computeScreenCoordinate: p
```

v14

"returns the fourteenth vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
^self computeScreenCoordinate: p
```

v15

"returns the fifteenth vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: (x + self
apronDepth value) withZ: (x + 5.5) negated.
^self computeScreenCoordinate: p
```

v16

"returns the sixteenth vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5)) withY: x withZ: (x + 5.5)
negated.
^self computeScreenCoordinate: p
```

v17

"returns the seventeenth vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
```

```

p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: (x + 5.5) negated.
^self computeScreenCoordinate: p

```

v18

"returns the eighteenth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: (x + self prosceniumHeight - 5.5).
^self computeScreenCoordinate: p

```

v19

"returns the nineteenth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (x +
self prosceniumHeight - 5.5).
^self computeScreenCoordinate: p

```

v2

"returns the second vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: (x + 9)
negated.
^self computeScreenCoordinate: p

```

v20

"returns the twentieth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (x +
5.5) negated.
^self computeScreenCoordinate: p

```

v21

"returns the twentyfirst vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: (x + 9) negated.
^self computeScreenCoordinate: p

```

v22

"returns the twentysecond vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) negated withY: x
withZ: (x + self prosceniumHeight - 2).
^self computeScreenCoordinate: p

```

v23

"returns the twentythird vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment1Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment1Depth)) withZ: (x + self roofSegment1Height).
^self computeScreenCoordinate: p

```

v24

"returns the twentyfourth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment2Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment2Depth)) withZ: (x + self roofSegment2Height).
^self computeScreenCoordinate: p

```

v25

"returns the twentyfifth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment3Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self
roofSegment3Depth)) withZ: (self roofSegment3Height + x).

```

^self computeScreenCoordinate: p

v26

"returns the twentysixth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self roofSegment4Depth) + 6) negated withY: (x + (self wallSplayAngle cos*self roofSegment4Depth)) withZ: (self roofSegment4Height + x).

^self computeScreenCoordinate: p

v27

"returns the twentyseventh vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (x + self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight).

^self computeScreenCoordinate: p

v28

"returns the twentyeighth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (x + self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight - 9).

^self computeScreenCoordinate: p

v29

"returns the twentyninth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) - (self wallSplayAngle sin*self balconyDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth) - (self wallSplayAngle cos*self balconyDepth)) withZ: (x + self seatingHeight + self balconyClearanceHeight - 9).

^self computeScreenCoordinate: p

v3

"returns the third vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: (x + 9) negated.

^self computeScreenCoordinate: p

v30

"returns the thirtieth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (x + self seatingHeight + self balconyClearanceHeight - 9).

^self computeScreenCoordinate: p

v31

"returns the thirtyfirst vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self auditoriumDepth) + 6) negated withY: (x + (self wallSplayAngle cos*self auditoriumDepth)) withZ: (x + self seatingHeight - 9).

^self computeScreenCoordinate: p

v32

"returns the thirtysecond vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle sin*self frontRowDistance) + 6) negated withY: (x + (self wallSplayAngle cos*self frontRowDistance)) withZ: (x + 9) negated.

^self computeScreenCoordinate: p

v33

"returns the thirtythird vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (x +
self prosceniumHeight - 2).
^self computeScreenCoordinate: p

```

v34

"returns the thirtyfourth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment1Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment1Depth)) withZ: (x + self roofSegment1Height).
^self computeScreenCoordinate: p

```

v35

"returns the thirtyfifth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment2Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment2Depth)) withZ: (x + self roofSegment2Height).
^self computeScreenCoordinate: p

```

v36

"returns the thirtysixth vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment3Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment3Depth)) withZ: (self roofSegment3Height + x).
^self computeScreenCoordinate: p

```

v37

"returns the thirtyseventh vertex of the auditorium as a screen coordinate"

```

| x p |
x := 0.000001.

```

```

p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self roofSegment4Depth) + 6) withY: (x + (self wallSplayAngle cos*self
roofSegment4Depth)) withZ: (self roofSegment4Height + x).
^self computeScreenCoordinate: p

```

v38

"returns the thirtieth vertex of the auditorium as a screen coordinate"

```

|x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (x + self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight).
^self computeScreenCoordinate: p

```

v39

"returns the thirtyninth vertex of the auditorium as a screen coordinate"

```

|x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (x + self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight -
9).
^self computeScreenCoordinate: p

```

v4

"returns the fourth vertex of the auditorium as a screen coordinate"

```

|x p |
x := 0.000001.
p := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth)
negated withZ: (x + 9) negated.
^self computeScreenCoordinate: p

```

v40

"returns the fortieth vertex of the auditorium as a screen coordinate"

```

|x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) - (self wallSplayAngle sin*self balconyDepth) + 6) withY: (x +

```

```
(self wallSplayAngle cos*self auditoriumDepth) -(self wallSplayAngle cos*self
balconyDepth)) withZ: (x + self seatingHeight + self balconyClearanceHeight - 9).
^self computeScreenCoordinate: p
```

v41

"returns the fortyfirst vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (x + self seatingHeight + self balconyClearanceHeight - 9).
^self computeScreenCoordinate: p
```

v42

"returns the fortysecond vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self auditoriumDepth) + 6) withY: (x + (self wallSplayAngle cos*self auditoriumDepth))
withZ: (x + self seatingHeight - 9).
^self computeScreenCoordinate: p
```

v43

"returns the fortythird vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + (self wallSplayAngle
sin*self frontRowDistance) + 6) withY: (x + (self wallSplayAngle cos*self
frontRowDistance)) withZ: (x + 9) negated.
^self computeScreenCoordinate: p
```

v44

"returns the fortyfourth vertex of the auditorium as a screen coordinate"

```
| x p |
x := 0.000001.
p := PointVector withX: (x + (self prosceniumWidth*0.5) + 6) withY: x withZ: (x +
9) negated.
^self computeScreenCoordinate: p
```

v45

"returns the fortyfifth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self frontRowDistance) withZ: (x + 9) negated.

^self computeScreenCoordinate: p

v46

"returns the fortysixth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (x + self seatingHeight - 9).

^self computeScreenCoordinate: p

v47

"returns the fortyseventh vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (x + self seatingHeight + self balconyClearanceHeight - 9).

^self computeScreenCoordinate: p

v48

"returns the fortyeighth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self auditoriumDepth - self balconyDepth) withZ: (x + self seatingHeight + self balconyClearanceHeight - 9).

^self computeScreenCoordinate: p

v49

"returns the fortyninth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (x + self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight - 9).

^self computeScreenCoordinate: p

v5

"returns the fifth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self stageWidth*0.5)) negated withY: (x + self stageDepth) negated withZ: ((x + self stageHeight) - 9).

^self computeScreenCoordinate: p

v50

"returns the fiftieth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self auditoriumDepth) withZ: (x + self seatingHeight + self balconyClearanceHeight + self balconySeatingHeight).

^self computeScreenCoordinate: p

v51

"returns the fiftyfirst vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self roofSegment4Depth) withZ: (x + self roofSegment4Height).

^self computeScreenCoordinate: p

v52

"returns the fiftysecond vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self roofSegment3Depth) withZ: (x + self roofSegment3Height).

^self computeScreenCoordinate: p

v53

"returns the fiftythird vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self roofSegment2Depth) withZ: (x + self roofSegment2Height).

^self computeScreenCoordinate: p

v54

"returns the fiftyfourth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: (x + self roofSegment1Depth) withZ: (x + self roofSegment1Height).

^self computeScreenCoordinate: p

v55

"returns the fiftyfifth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: x withY: x withZ: (x + self prosceniumHeight - 2).

^self computeScreenCoordinate: p

v6

"returns the sixth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self stageWidth*0.5)) negated withY: x withZ: ((x + self stageHeight) - 9).

^self computeScreenCoordinate: p

v7

"returns the seventh vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self stageWidth*0.5)) withY: x withZ: ((x + self stageHeight) - 9).

^self computeScreenCoordinate: p

v8

"returns the eighth vertex of the auditorium as a screen coordinate"

| x p |

x := 0.000001.

p := PointVector withX: (x + (self stageWidth*0.5)) withY: (x + self stageDepth)
 negated withZ: ((x + self stageHeight) - 9).
 ^self computeScreenCoordinate: p

v9

"returns the ninth vertex of the auditorium as a screen coordinate"

| x p |
 x := 0.000001.
 p := PointVector withX: (x + (self prosceniumWidth*0.5)) negated withY: x withZ:
 (x + 9) negated.
 ^self computeScreenCoordinate: p

wallSplayAngle

"returns the splay angle (in radians) of the side walls of the auditorium after it has been
 optimized for visual comfort"

| x y z |
 x := ((self wallSplayAngleBasedOnSeatingArea) min: 30) degreesToRadians.
 y := x min: self wallSplayAngleFromIacc.
 z := y min: self wallSplayAngleFromTrebleRatio.
 ^z

wallSplayAngleBasedOnSeatingArea

"calculates and returns the splay angle (in degrees) of the side walls of the auditorium
 based on seating area"

^(60*self seatingArea)/(3.142*((self auditoriumDepth - self frontRowDistance)
 squared))

wallSplayAngleFromIacc

"returns the splay angle (in radians) of the side walls of the auditorium based on the
 inter aural cross correlation parameter"

^((self iacc value - 0.284)/(0.005*self auditoriumDepth)) arcTan abs

wallSplayAngleFromTrebleRatio

"returns the splay angle (in radians) of the side walls of the auditorium based on the
 treble ratio parameter"

^((self trebleRatio value - 0.949)/(0.002*self auditoriumDepth)) negated arcTan abs

Auditorium methodsFor: 'computing'

computeScreenCoordinate: aPointVector

"computes the screen coordinates of a point vector"

```
| transformedPointVector screenCoordinate |
transformedPointVector := self transMatrix multiply4: aPointVector.
screenCoordinate := transformedPointVector extractPointWith: self
viewingPlaneDistance value.
^screenCoordinate
```

Auditorium methodsFor: 'planesProcessing'

setColoredPlanes

"sets the colors as the final stage in setting all the parameters of the screen planes that make up the image of the auditorium and returns the planes ready for display"

```
| z |
z := self setSortedPlanesNormalized.
z do: [:each | | x y m n |
x := (self lightpoint latitude - ((each zNormal) arcCos))
radiansToDegrees.
y := (self lightpoint longitude - ((each xNormal)
arcCos)) radiansToDegrees.
m := self eyepoint latitude radiansToDegrees - x.
n := self eyepoint longitude radiansToDegrees -
y.
each color: (ColorValue
hue: 0.20
saturation: 1.0
brightness: (((1 - m degreesToRadians cos abs) + (n degreesToRadians
cos abs))*0.5)).
^z
```

setSortedPlanesNormalized

"sets and returns the screen planes of the auditorium with their normal components and distance from the origin computed, and sorted in the proper order for display"

```
| x z |
x := self planes do: [:each | each transformUsing: (self transMatrix)].
z := SortedCollection sortBlock: [:p :q | | m n i |
m := p maximumZ.
n := q maximumZ.
i := 1.
[(((m at: i) z) = ((n at: i) z)) and: [(i < m size) & (i < n size)]]
```

```

        whileTrue: [i := i + 1].
        ((m at: i) z) > ((n at: i) z)].
    z addAll: x.
    ^z

```

Auditorium methodsFor: 'initializing'

initialize

"initializes the instance variables of an auditorium"

```

    eyepointDistance := 500 asValue.
    self eyepointDistance onChangeSend: #setEyepoint to: self.
    eyepointLatitude := 45 asValue.
    self eyepointLatitude onChangeSend: #setEyepoint to: self.
    eyepointLongitude := 60 asValue.
    self eyepointLongitude onChangeSend: #setEyepoint to: self.
    lightpointDistance := 300 asValue.
    self lightpointDistance onChangeSend: #setLightpoint to: self.
    lightpointLatitude := 45 asValue.
    self lightpointLatitude onChangeSend: #setLightpoint to: self.
    lightpointLongitude := 60 asValue.
    self lightpointLongitude onChangeSend: #setLightpoint to: self.
    eyepoint := ((EyePoint new) distance: self eyepointDistance value latitude: self
    eyepointLatitude value longitude: self eyepointLongitude value).
    lightpoint := ((LightPoint new) distance: self lightpointDistance value latitude: self
    lightpointLatitude value longitude: self lightpointLongitude value).
    viewingPlaneDistance := 90 asValue.
    self viewingPlaneDistance onChangeSend: #setPlanes to: self.
    auditoriumCapacity := 2100 asValue.
    self auditoriumCapacity onChangeSend: #setDataReportAndPlanes to: self.
    areaPerSeat := 6.5 asValue.
    self areaPerSeat onChangeSend: #setDataReportAndPlanes to: self.
    apronDepth := 8 asValue.
    self apronDepth onChangeSend: #setDataReportAndPlanes to: self.
    auditoriumDepthFromVisualClarity := 120 asValue.
    self auditoriumDepthFromVisualClarity onChangeSend: #setDataReportAndPlanes
to: self.
    seatingSlope := 20 asValue.
    self seatingSlope onChangeSend: #setDataReportAndPlanes to: self.
    performanceMode := 'drama' asValue.
    self performanceMode onChangeSend: #setStageDimensionsReportAndPlanes to:
self.
    loudnessLossAllowable := 4 asValue.

```

```

self loudnessLossAllowable onChangeSend: #setDataReportAndPlanes to: self.
reverberationTime := 2.5 asValue.
self reverberationTime onChangeSend: #setDataReportAndPlanes to: self.
timeDelay1 := 0.04 asValue.
self timeDelay1 onChangeSend: #setDataReportAndPlanes to: self.
timeDelay2 := 0.043 asValue.
self timeDelay2 onChangeSend: #setDataReportAndPlanes to: self.
timeDelay3 := 0.046 asValue.
self timeDelay3 onChangeSend: #setDataReportAndPlanes to: self.
timeDelay4 := 0.049 asValue.
self timeDelay4 onChangeSend: #setDataReportAndPlanes to: self.
iacc := 0.64 asValue.
self iacc onChangeSend: #setDataReportAndPlanes to: self.
trebleRatio := 0.67 asValue.
self trebleRatio onChangeSend: #setDataReportAndPlanes to: self.
self setStageDimensionsAndPlanes.
dataReport := self compileDataReport asValue.
planeView := (AuditoriumPlaneView new model: self).
frameView := (AuditoriumFrameView new model: self).

```

Auditorium methodsFor: 'aspects'

apronDepth

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

```
^apronDepth isNil ifTrue: [apronDepth := 2 asValue] ifFalse: [apronDepth]
```

areaPerSeat

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

```
^areaPerSeat isNil ifTrue: [areaPerSeat := 5 asValue] ifFalse: [areaPerSeat]
```

auditoriumCapacity

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

```
^auditoriumCapacity isNil ifTrue: [auditoriumCapacity := 500 asValue] ifFalse:
[auditoriumCapacity]
```

auditoriumDepthFromVisualClarity

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^auditoriumDepthFromVisualClarity isNil ifTrue: [auditoriumDepthFromVisualClarity := 1 asValue] ifFalse: [auditoriumDepthFromVisualClarity]`

`dataReport`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^dataReport isNil ifTrue: [dataReport := String new asValue] ifFalse: [dataReport]`

`eyepointDistance`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^eyepointDistance isNil ifTrue: [eyepointDistance := 1 asValue] ifFalse: [eyepointDistance]`

`eyepointLatitude`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^eyepointLatitude isNil ifTrue: [eyepointLatitude := 1 asValue] ifFalse: [eyepointLatitude]`

`eyepointLongitude`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^eyepointLongitude isNil ifTrue: [eyepointLongitude := 1 asValue] ifFalse: [eyepointLongitude]`

`iacc`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^iacc isNil ifTrue: [iacc := 0.01 asValue] ifFalse: [iacc]`

`lightpointDistance`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^lightpointDistance isNil ifTrue: [lightpointDistance := 1 asValue] ifFalse:
[lightpointDistance]`

`lightpointLatitude`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^lightpointLatitude isNil ifTrue: [lightpointLatitude := 1 asValue] ifFalse:
[lightpointLatitude]`

`lightpointLongitude`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^lightpointLongitude isNil ifTrue: [lightpointLongitude := 1 asValue] ifFalse:
[lightpointLongitude]`

`loudnessLossAllowable`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^loudnessLossAllowable isNil ifTrue: [loudnessLossAllowable := 3 asValue] ifFalse:
[loudnessLossAllowable]`

`performanceMode`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^performanceMode isNil ifTrue: [performanceMode := String new asValue] ifFalse:
[performanceMode]`

`reverberationTime`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^reverberationTime isNil ifTrue: [reverberationTime := 0.8 asValue] ifFalse:
[reverberationTime]`

`seatingSlope`

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

`^seatingSlope isNil ifTrue: [seatingSlope := 0.0 asValue] ifFalse: [seatingSlope]`

timeDelay1

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

^timeDelay1 isNil ifTrue: [timeDelay1 := 1 asValue] ifFalse: [timeDelay1]

timeDelay2

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

^timeDelay2 isNil ifTrue: [timeDelay2 := 1 asValue] ifFalse: [timeDelay2]

timeDelay3

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

^timeDelay3 isNil ifTrue: [timeDelay3 := 1 asValue] ifFalse: [timeDelay3]

timeDelay4

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

^timeDelay4 isNil ifTrue: [timeDelay4 := 1 asValue] ifFalse: [timeDelay4]

trebleRatio

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

^trebleRatio isNil ifTrue: [trebleRatio := 0.01 asValue] ifFalse: [trebleRatio]

viewingPlaneDistance

"This method was generated by UIDefiner. The initialization provided below may have been preempted by an initialize method."

^viewingPlaneDistance isNil ifTrue: [viewingPlaneDistance := 1 asValue] ifFalse: [viewingPlaneDistance]

"-----"

Auditorium class

instanceVariableNames: "

Auditorium class methodsFor: 'instance creation'

new

"creates an instance of an auditorium and initializes its variables"

^super new initialize

Auditorium class methodsFor: 'interface specs'

windowSpec

"UIPainter new openOnClass: self andSelector: #windowSpec"

```
^((FullSpec #window: ((WindowSpec #label: 'Auditorium Model' #min: ((Point
640 480 ) #bounds: ((Rectangle 144 23 784 503 ) ) #component: ((SpecCollection
#collection: ((InputFieldSpec #layout: ((LayoutFrame 0 0.0140625 0 0.966667 0
0.0859375 0 1.0 ) #model: #lightpointDistance #isReadOnly: true #type: #number )
#ArbitraryComponentSpec #layout: ((LayoutFrame 0 0.617187 0 0.65 0 0.992187 0
0.983333 ) #component: #planeView ) #SliderSpec #layout: ((LayoutFrame 0 0.0984375
0 0.966667 0 0.329688 0 1.0 ) #model: #lightpointDistance #orientation: #horizontal #start:
1 #stop: 1000 #step: 1 ) #InputFieldSpec #layout: ((LayoutFrame 0 0.0140625 0
0.916667 0 0.0859375 0 0.95 ) #model: #lightpointLongitude #isReadOnly: true #type:
#number ) #SliderSpec #layout: ((LayoutFrame 0 0.0984375 0 0.916667 0 0.329688 0
0.95 ) #model: #lightpointLongitude #orientation: #horizontal #start: 1 #stop: 360 #step: 1
) #InputFieldSpec #layout: ((LayoutFrame 0 0.0140625 0 0.766667 0 0.0859375 0 0.8
) #model: #eyepointLongitude #isReadOnly: true #type: #number ) #SliderSpec #layout:
((LayoutFrame 0 0.0984375 0 0.766667 0 0.329688 0 0.8 ) #model: #eyepointLongitude
#orientation: #horizontal #start: 1 #stop: 360 #step: 1 ) #InputFieldSpec #layout:
((LayoutFrame 0 0.0140625 0 0.716667 0 0.0859375 0 0.75 ) #model: #eyepointLatitude
#isReadOnly: true #type: #number ) #SliderSpec #layout: ((LayoutFrame 0 0.0984375
0 0.716667 0 0.329688 0 0.75 ) #model: #eyepointLatitude #orientation: #horizontal #start:
1 #stop: 90 #step: 1 ) #InputFieldSpec #layout: ((LayoutFrame 0 0.0140625 0 0.266667
0 0.0859375 0 0.3 ) #model: #loudnessLossAllowable #isReadOnly: true #type: #number )
#SliderSpec #layout: ((LayoutFrame 0 0.0984375 0 0.26875 0 0.329688 0 0.3 ) #model:
#loudnessLossAllowable #orientation: #horizontal #start: 3 #stop: 8 #step: 0.5 )
#InputFieldSpec #layout: ((LayoutFrame 0 0.0140625 0 0.0666667 0 0.0859375 0 0.1
) #model: #areaPerSeat #isReadOnly: true #type: #number ) #SliderSpec #layout:
((LayoutFrame 0 0.0984375 0 0.0666667 0 0.329688 0 0.1 ) #model: #areaPerSeat
#orientation: #horizontal #start: 4 #stop: 8 #step: 0.2 ) #InputFieldSpec #layout:
((LayoutFrame 0 0.0140625 0 0.116667 0 0.084375 0 0.15 ) #model: #apronDepth
#isReadOnly: true #type: #number ) #SliderSpec #layout: ((LayoutFrame 0 0.0984375
0 0.114583 0 0.329688 0 0.15 ) #model: #apronDepth #orientation: #horizontal #start: 5
#stop: 20 #step: 0.5 ) #InputFieldSpec #layout: ((LayoutFrame 0 0.0140625 0 0.0166667
0 0.0859375 0 0.05 ) #model: #auditoriumCapacity #isReadOnly: true #type: #number )
#SliderSpec #layout: ((LayoutFrame 0 0.0984375 0 0.0125 0 0.329688 0 0.05 ) #model:
#auditoriumCapacity #orientation: #horizontal #start: 500 #stop: 3000 #step: 5 )
```

```

##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.0140625 0 0.866667 0 0.0859375 0 0.9 )
#model: #lightpointLatitude #isReadOnly: true #type: #number ) ##(#SliderSpec #layout:
##(#LayoutFrame 0 0.0984375 0 0.866667 0 0.329688 0 0.9 ) #model: #lightpointLatitude
#orientation: #horizontal #start: 1 #stop: 90 #step: 1 ) ##(#InputFieldSpec #layout:
##(#LayoutFrame 0 0.0140625 0 0.816667 0 0.0859375 0 0.85 ) #model: #eyepointDistance
#isReadOnly: true #type: #number ) ##(#SliderSpec #layout: ##(#LayoutFrame 0 0.0984375
0 0.816667 0 0.329688 0 0.85 ) #model: #eyepointDistance #orientation: #horizontal #start:
1 #stop: 1000 #step: 1 ) ##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.539062 0 0.483333
0 0.6875 0 0.533333 ) #model: #performanceMode #menu: #modeMenu #type: #string )
##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.0140625 0 0.166667 0 0.0859375 0 0.2 )
#model: #auditoriumDepthFromVisualClarity #isReadOnly: true #type: #number )
##(#SliderSpec #layout: ##(#LayoutFrame 0 0.0984375 0 0.166667 0 0.329688 0 0.2 ) #model:
#auditoriumDepthFromVisualClarity #orientation: #horizontal #start: 80 #stop: 135 #step:
1 ) ##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.0140625 0 0.366667 0 0.0859375 0 0.4
) #model: #timeDelay2 #isReadOnly: true #type: #number ) ##(#SliderSpec #layout:
##(#LayoutFrame 0 0.0984375 0 0.366667 0 0.329688 0 0.4 ) #model: #timeDelay2
#orientation: #horizontal #start: 0.02 #stop: 0.08 #step: 0.002 ) ##(#InputFieldSpec #layout:
##(#LayoutFrame 0 0.0140625 0 0.316667 0 0.0859375 0 0.35 ) #model: #timeDelay1
#isReadOnly: true #type: #number ) ##(#SliderSpec #layout: ##(#LayoutFrame 0 0.0984375
0 0.316667 0 0.329688 0 0.35 ) #model: #timeDelay1 #orientation: #horizontal #start: 0.02
#stop: 0.08 #step: 0.002 ) ##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.0140625 0
0.466667 0 0.0859375 0 0.5 ) #model: #timeDelay4 #isReadOnly: true #type: #number )
##(#SliderSpec #layout: ##(#LayoutFrame 0 0.0984375 0 0.466667 0 0.329688 0 0.5 ) #model:
#timeDelay4 #orientation: #horizontal #start: 0.02 #stop: 0.08 #step: 0.002 )
##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.0140625 0 0.416667 0 0.0859375 0 0.45
) #model: #timeDelay3 #isReadOnly: true #type: #number ) ##(#SliderSpec #layout:
##(#LayoutFrame 0 0.0984375 0 0.416667 0 0.329688 0 0.45 ) #model: #timeDelay3
#orientation: #horizontal #start: 0.02 #stop: 0.08 #step: 0.002 ) ##(#InputFieldSpec #layout:
##(#LayoutFrame 0 0.0140625 0 0.666667 0 0.0859375 0 0.7 ) #model:
#viewingPlaneDistance #isReadOnly: true #type: #number ) ##(#SliderSpec #layout:
##(#LayoutFrame 0 0.0984375 0 0.666667 0 0.329688 0 0.7 ) #model: #viewingPlaneDistance
#orientation: #horizontal #start: 1 #stop: 1000 #step: 1 ) ##(#ArbitraryComponentSpec
#layout: ##(#LayoutFrame 0 0.617187 0 0.0166667 0 0.992187 0 0.366667 ) #component:
#frameView ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.539062 0 0.433333 ) #isOpaque:
true #label: 'Performance Mode' ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0
0.666667 ) #isOpaque: true #label: 'Viewing Plane Distance (ft)' ) ##(#LabelSpec #layout:
##(#LayoutOrigin 0 0.339063 0 0.716667 ) #isOpaque: true #label: 'Eyepoint Latitude (deg)'
) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.766667 ) #isOpaque: true #label:
'Eyepoint Longitude (deg)' ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.816667
) #isOpaque: true #label: 'Eyepoint Distance (ft)' ) ##(#LabelSpec #layout: ##(#LayoutOrigin
0 0.339063 0 0.866667 ) #isOpaque: true #label: 'Lightpoint Latitude (deg)' ) ##(#LabelSpec
#layout: ##(#LayoutOrigin 0 0.339063 0 0.916667 ) #isOpaque: true #label: 'Lightpoint
Longitude (deg)' ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.966667 )

```

```

#isOpaque: true #label: 'Lightpoint Distance (ft)' ) ##(#LabelSpec #layout: ##(#LayoutOrigin
0 0.339063 0 0.0166667 ) #isOpaque: true #label: 'Auditorium Capacity' ) ##(#LabelSpec
#layout: ##(#LayoutOrigin 0 0.339063 0 0.116667 ) #isOpaque: true #label: 'Apron Depth (ft)'
) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.266667 ) #isOpaque: true #label:
'dB Loss Allowable' ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.166667 )
#isOpaque: true #label: 'Depth for Visual Clarity (ft)' ) ##(#LabelSpec #layout:
##(#LayoutOrigin 0 0.339063 0 0.316667 ) #isOpaque: true #label: 'Time Delay 1 (sec)' )
##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.366667 ) #isOpaque: true #label:
'Time Delay 2 (sec)' ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.416667 )
#isOpaque: true #label: 'Time Delay 3 (sec)' ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0
0.339063 0 0.0666667 ) #isOpaque: true #label: 'Area/Seat (sft.)' ) ##(#LabelSpec #layout:
##(#LayoutOrigin 0 0.339063 0 0.466667 ) #isOpaque: true #label: 'Time Delay 4 (sec)' )
##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.83125 0 0.383333 ) #isOpaque: true #label:
'Wire-frame Image' ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.803125 0 0.597917 )
#isOpaque: true #label: 'Shaded Plane Image' ) ##(#InputFieldSpec #layout: ##(#LayoutFrame
0 0.0140625 0 0.616667 0 0.0875 0 0.65 ) #model: #reverberationTime #isReadOnly: true
#type: #number ) ##(#SliderSpec #layout: ##(#LayoutFrame 0 0.0984375 0 0.616667 0
0.329688 0 0.65 ) #model: #reverberationTime #orientation: #horizontal #start: 0.8 #stop:
2.5 #step: 0.1 ) ##(#LabelSpec #layout: ##(#LayoutOrigin 0 0.339063 0 0.616667 ) #isOpaque:
true #label: 'RT (sec)' ) ##(#TextEditorSpec #layout: ##(#LayoutFrame 0 0.717187 0 0.433333
0 0.992187 0 0.583333 ) #model: #dataReport #isReadOnly: true ) ##(#InputFieldSpec
#layout: ##(#LayoutFrame 0 0.0140625 0 0.516667 0 0.0859375 0 0.55 ) #model: #iacc
#isReadOnly: true #type: #number ) ##(#SliderSpec #layout: ##(#LayoutFrame 0 0.0984375
0 0.516667 0 0.329688 0 0.55 ) #model: #iacc #orientation: #horizontal #start: 0.01 #stop:
1.0 #step: 0.01 ) ##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.0140625 0 0.566667 0
0.0859375 0 0.6 ) #model: #trebleRatio #isReadOnly: true #type: #number ) ##(#SliderSpec
#layout: ##(#LayoutFrame 0 0.0984375 0 0.566667 0 0.329688 0 0.6 ) #model: #trebleRatio
#orientation: #horizontal #start: 0.01 #stop: 1.2 #step: 0.01 ) ##(#LabelSpec #layout:
##(#LayoutOrigin 0 0.339063 0 0.516667 ) #isOpaque: true #label: 'IACC' ) ##(#LabelSpec
#layout: ##(#LayoutOrigin 0 0.339063 0 0.566667 ) #isOpaque: true #label: 'Treble Ratio' )
##(#InputFieldSpec #layout: ##(#LayoutFrame 0 0.0140625 0 0.216667 0 0.0875 0 0.25 )
#model: #seatingSlope #isReadOnly: true #type: #number ) ##(#SliderSpec #layout:
##(#LayoutFrame 0 0.0984375 0 0.216667 0 0.329688 0 0.25 ) #model: #seatingSlope
#orientation: #horizontal #start: 0.0 #stop: 60.0 #step: 0.5 ) ##(#LabelSpec #layout:
##(#LayoutOrigin 0 0.339063 0 0.216667 ) #isOpaque: true #label: 'Seating Slope (deg)' )
) )

```

Auditorium class methodsFor: 'resources'

modeMenu

"UIMenuEditor new openOnClass: self andSelector: #modeMenu"

```
^#(#PopupMenu #('Theater' 'Drama' 'Musical' 'Symphony' 'Opera' ) #() #(#setTheater  
#setDrama #setMusical #setSymphony #setOpera ) ) decodeAsLiteralArray
```

```

Auditorium subclass: #RectangularAuditorium
    instanceVariableNames: "
    classVariableNames: "
    poolDictionaries: "
    category: 'Auditorium'

```

RectangularAuditorium methodsFor: 'accessing'

approximateWallAndRoofSurfaceArea

"returns the approximate wall and roof surface area of the auditorium assuming flat roof segments and neglecting the strip area around the proscenium"

```

| p q r s t u surfaceArea |
p := (self prosceniumWidth + 12)*(self wallSplayAngle cos*self auditoriumDepth).
q := (self wallSplayAngle cos + self wallSplayAngle sin)*self auditoriumDepth.
r := ((self prosceniumWidth*0.5) + 6 + (self wallSplayAngle sin*self
auditoriumDepth))*(self auditoriumDepth - (self wallSplayAngle cos*self auditoriumDepth)).
s := self prosceniumWidth + 12.
t := (self balconyClearanceHeight + 9)*s.
u := self averageAuditoriumHeight*self auditoriumDepth*2.
surfaceArea := (p + q + r + t + u).
^surfaceArea

```

averageAuditoriumWidth

"returns the average width of the auditorium based on a fan shape type equivalent"

```

| offset |
offset := self auditoriumDepth*(super wallSplayAngle sin).
^(prosceniumWidth + offset)

```

averageWallAbsorptionCoefficient

"returns the average absorption coefficient for materials to be used on wall surfaces in the auditorium"

```

| s t u wallSurfaceArea |
s := self prosceniumWidth + 12.
t := (self balconyClearanceHeight + 9)*s.
u := self averageAuditoriumHeight*self auditoriumDepth*2.
wallSurfaceArea := t + u.
^self roomConstant/wallSurfaceArea

```

balconyArea

"returns the balcony area of the auditorium adjusted for constraints"

$(\text{self seatingArea} - ((\text{self prosceniumWidth} + 12) * \text{self auditoriumDepth})) > 0.0$

if True: $[(\text{self seatingArea} - ((\text{self prosceniumWidth} + 12) * \text{self auditoriumDepth}))$

min: $(\text{self seatingArea} * 0.2)]$

if False: $[\text{0.0}]$

balconyDepth

"returns the balcony depth of the auditorium with a depth restriction of 0.25 times the depth of the auditorium"

$\wedge(\text{self balconyArea} / (\text{self prosceniumWidth} + 12))$ min: $(\text{self auditoriumDepth} * 0.25)$

balconySeatingHeight

"returns the balcony seating height of the auditorium"

$\text{self balconyArea} > 0.0$

if True: $[\text{self balconyDepth} * 0.577]$

if False: $[\text{0.0}]$

balconyShortfall

"returns the percentage of the seating area shortfall due to the balcony area and depth constraints"

$\wedge(((\text{self seatingArea} - ((\text{self prosceniumWidth} + 12) * \text{self auditoriumDepth}) - ((\text{self prosceniumWidth} + 12) * \text{self balconyDepth})) / \text{self seatingArea}) * 100)$ max: 0.0

prosceniumWidth

"returns the width of the proscenium of the auditorium adjusted for conversion from the fan to rectangular shape type"

$| \text{offset} |$

$\text{offset} := \text{self auditoriumDepth} * (\text{super wallSplayAngle} \sin).$

$\wedge(\text{prosceniumWidth} + \text{offset})$

wallSplayAngle

"returns the splay angle (in radians) of the side walls of the auditorium"

$\wedge 0$ degreesToRadians

RectangularAuditorium methodsFor: 'setting'

setStageDimensions

"sets the stage dimensions of the auditorium based on standards adjusted for conversion from the fan to rectangular shape type"

```
| offset |
offset := self auditoriumDepth*(super wallSplayAngle sin).
stageDepth := (self prosceniumWidth - offset)*1.25.
stageHeight := (self prosceniumHeight*2.75) + 9.
stageWidth := (self prosceniumWidth - offset)*2.5
"-----"
```

RectangularAuditorium class
instanceVariableNames: "

RectangularAuditorium class methodsFor: 'interface specs'

windowSpec

"UIPainter new openOnClass: self andSelector: #windowSpec"

```
^(#FullSpec #window: (#WindowSpec #label: 'Auditorium Model' #min: (#Point
640 480 ) #bounds: (#Rectangle 144 23 784 503 ) ) #component: (#SpecCollection
#collection: (#(#InputFieldSpec #layout: (#LayoutFrame 0 0.0140625 0 0.966667 0
0.0859375 0 1.0 ) #model: #lightpointDistance #isReadOnly: true #type: #number )
#(#ArbitraryComponentSpec #layout: (#LayoutFrame 0 0.617187 0 0.65 0 0.992187 0
0.983333 ) #component: #planeView ) #(#SliderSpec #layout: (#LayoutFrame 0 0.0984375
0 0.966667 0 0.329688 0 1.0 ) #model: #lightpointDistance #orientation: #horizontal #start:
1 #stop: 1000 #step: 1 ) #(#InputFieldSpec #layout: (#LayoutFrame 0 0.0140625 0
0.916667 0 0.0859375 0 0.95 ) #model: #lightpointLongitude #isReadOnly: true #type:
#number ) #(#SliderSpec #layout: (#LayoutFrame 0 0.0984375 0 0.916667 0 0.329688 0
0.95 ) #model: #lightpointLongitude #orientation: #horizontal #start: 1 #stop: 360 #step: 1
) #(#InputFieldSpec #layout: (#LayoutFrame 0 0.0140625 0 0.766667 0 0.0859375 0 0.8
) #model: #eyepointLongitude #isReadOnly: true #type: #number ) #(#SliderSpec #layout:
#LayoutFrame 0 0.0984375 0 0.766667 0 0.329688 0 0.8 ) #model: #eyepointLongitude
#orientation: #horizontal #start: 1 #stop: 360 #step: 1 ) #(#InputFieldSpec #layout:
#LayoutFrame 0 0.0140625 0 0.716667 0 0.0859375 0 0.75 ) #model: #eyepointLatitude
#isReadOnly: true #type: #number ) #(#SliderSpec #layout: (#LayoutFrame 0 0.0984375
0 0.716667 0 0.329688 0 0.75 ) #model: #eyepointLatitude #orientation: #horizontal #start:
1 #stop: 90 #step: 1 ) #(#InputFieldSpec #layout: (#LayoutFrame 0 0.0140625 0 0.266667
0 0.0859375 0 0.3 ) #model: #loudnessLossAllowable #isReadOnly: true #type: #number )
#(#SliderSpec #layout: (#LayoutFrame 0 0.0984375 0 0.26875 0 0.329688 0 0.3 ) #model:
#loudnessLossAllowable #orientation: #horizontal #start: 3 #stop: 8 #step: 0.5 )
#(#InputFieldSpec #layout: (#LayoutFrame 0 0.0140625 0 0.066667 0 0.0859375 0 0.1
```

```

) #model: #areaPerSeat #isReadOnly: true #type: #number ) ##(SliderSpec #layout:
##(LayoutFrame 0 0.0984375 0 0.0666667 0 0.329688 0 0.1 ) #model: #areaPerSeat
#orientation: #horizontal #start: 4 #stop: 8 #step: 0.2 ) ##(InputFieldSpec #layout:
##(LayoutFrame 0 0.0140625 0 0.116667 0 0.084375 0 0.15 ) #model: #apronDepth
#isReadOnly: true #type: #number ) ##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375
0 0.114583 0 0.329688 0 0.15 ) #model: #apronDepth #orientation: #horizontal #start: 5
#stop: 20 #step: 0.5 ) ##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0 0.0166667
0 0.0859375 0 0.05 ) #model: #auditoriumCapacity #isReadOnly: true #type: #number )
##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375 0 0.0125 0 0.329688 0 0.05 ) #model:
#auditoriumCapacity #orientation: #horizontal #start: 500 #stop: 3000 #step: 5 )
##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0 0.866667 0 0.0859375 0 0.9 )
#model: #lightpointLatitude #isReadOnly: true #type: #number ) ##(SliderSpec #layout:
##(LayoutFrame 0 0.0984375 0 0.866667 0 0.329688 0 0.9 ) #model: #lightpointLatitude
#orientation: #horizontal #start: 1 #stop: 90 #step: 1 ) ##(InputFieldSpec #layout:
##(LayoutFrame 0 0.0140625 0 0.816667 0 0.0859375 0 0.85 ) #model: #eyepointDistance
#isReadOnly: true #type: #number ) ##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375
0 0.816667 0 0.329688 0 0.85 ) #model: #eyepointDistance #orientation: #horizontal #start:
1 #stop: 1000 #step: 1 ) ##(InputFieldSpec #layout: ##(LayoutFrame 0 0.539062 0 0.483333
0 0.6875 0 0.533333 ) #model: #performanceMode #menu: #modeMenu #type: #string )
##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0 0.166667 0 0.0859375 0 0.2 )
#model: #auditoriumDepthFromVisualClarity #isReadOnly: true #type: #number )
##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375 0 0.166667 0 0.329688 0 0.2 ) #model:
#auditoriumDepthFromVisualClarity #orientation: #horizontal #start: 80 #stop: 135 #step:
1 ) ##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0 0.366667 0 0.0859375 0 0.4
) #model: #timeDelay2 #isReadOnly: true #type: #number ) ##(SliderSpec #layout:
##(LayoutFrame 0 0.0984375 0 0.366667 0 0.329688 0 0.4 ) #model: #timeDelay2
#orientation: #horizontal #start: 0.02 #stop: 0.08 #step: 0.002 ) ##(InputFieldSpec #layout:
##(LayoutFrame 0 0.0140625 0 0.316667 0 0.0859375 0 0.35 ) #model: #timeDelay1
#isReadOnly: true #type: #number ) ##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375
0 0.316667 0 0.329688 0 0.35 ) #model: #timeDelay1 #orientation: #horizontal #start: 0.02
#stop: 0.08 #step: 0.002 ) ##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0
0.466667 0 0.0859375 0 0.5 ) #model: #timeDelay4 #isReadOnly: true #type: #number )
##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375 0 0.466667 0 0.329688 0 0.5 ) #model:
#timeDelay4 #orientation: #horizontal #start: 0.02 #stop: 0.08 #step: 0.002 )
##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0 0.416667 0 0.0859375 0 0.45
) #model: #timeDelay3 #isReadOnly: true #type: #number ) ##(SliderSpec #layout:
##(LayoutFrame 0 0.0984375 0 0.416667 0 0.329688 0 0.45 ) #model: #timeDelay3
#orientation: #horizontal #start: 0.02 #stop: 0.08 #step: 0.002 ) ##(InputFieldSpec #layout:
##(LayoutFrame 0 0.0140625 0 0.666667 0 0.0859375 0 0.7 ) #model:
#viewingPlaneDistance #isReadOnly: true #type: #number ) ##(SliderSpec #layout:
##(LayoutFrame 0 0.0984375 0 0.666667 0 0.329688 0 0.7 ) #model: #viewingPlaneDistance
#orientation: #horizontal #start: 1 #stop: 1000 #step: 1 ) ##(ArbitraryComponentSpec
#layout: ##(LayoutFrame 0 0.617187 0 0.0166667 0 0.992187 0 0.366667 ) #component:

```

```

#frameView ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.539062 0 0.433333 ) #isOpaque:
true #label: 'Performance Mode' ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0
0.666667 ) #isOpaque: true #label: 'Viewing Plane Distance (ft)' ) ##(LabelSpec #layout:
##(LayoutOrigin 0 0.339063 0 0.716667 ) #isOpaque: true #label: 'Eyepoint Latitude (deg)'
) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.766667 ) #isOpaque: true #label:
'Eyepoint Longitude (deg)' ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.816667
) #isOpaque: true #label: 'Eyepoint Distance (ft)' ) ##(LabelSpec #layout: ##(LayoutOrigin
0 0.339063 0 0.866667 ) #isOpaque: true #label: 'Lightpoint Latitude (deg)' ) ##(LabelSpec
#layout: ##(LayoutOrigin 0 0.339063 0 0.916667 ) #isOpaque: true #label: 'Lightpoint
Longitude (deg)' ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.966667 )
#isOpaque: true #label: 'Lightpoint Distance (ft)' ) ##(LabelSpec #layout: ##(LayoutOrigin
0 0.339063 0 0.0166667 ) #isOpaque: true #label: 'Auditorium Capacity' ) ##(LabelSpec
#layout: ##(LayoutOrigin 0 0.339063 0 0.116667 ) #isOpaque: true #label: 'Apron Depth (ft)'
) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.266667 ) #isOpaque: true #label:
'dB Loss Allowable' ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.166667 )
#isOpaque: true #label: 'Depth for Visual Clarity (ft)' ) ##(LabelSpec #layout:
##(LayoutOrigin 0 0.339063 0 0.316667 ) #isOpaque: true #label: 'Time Delay 1 (sec)' )
##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.366667 ) #isOpaque: true #label:
'Time Delay 2 (sec)' ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.416667 )
#isOpaque: true #label: 'Time Delay 3 (sec)' ) ##(LabelSpec #layout: ##(LayoutOrigin 0
0.339063 0 0.0666667 ) #isOpaque: true #label: 'Area/Seat (sq ft)' ) ##(LabelSpec #layout:
##(LayoutOrigin 0 0.339063 0 0.466667 ) #isOpaque: true #label: 'Time Delay 4 (sec)' )
##(LabelSpec #layout: ##(LayoutOrigin 0 0.83125 0 0.383333 ) #isOpaque: true #label:
'Wire-frame Image' ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.803125 0 0.597917 )
#isOpaque: true #label: 'Shaded Plane Image' ) ##(InputFieldSpec #layout: ##(LayoutFrame
0 0.0140625 0 0.616667 0 0.0875 0 0.65 ) #model: #reverberationTime #isReadOnly: true
#type: #number ) ##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375 0 0.616667 0
0.329688 0 0.65 ) #model: #reverberationTime #orientation: #horizontal #start: 0.8 #stop:
2.5 #step: 0.1 ) ##(LabelSpec #layout: ##(LayoutOrigin 0 0.339063 0 0.616667 ) #isOpaque:
true #label: 'RT (sec)' ) ##(TextEditorSpec #layout: ##(LayoutFrame 0 0.717187 0 0.433333
0 0.992187 0 0.583333 ) #model: #dataReport #isReadOnly: true ) ##(InputFieldSpec
#layout: ##(LayoutFrame 0 0.0140625 0 0.516667 0 0.0859375 0 0.55 ) #model: #iacc
#isReadOnly: true #type: #number ) ##(SliderSpec #layout: ##(LayoutFrame 0 0.0984375
0 0.516667 0 0.329688 0 0.55 ) #model: #iacc #orientation: #horizontal #start: 0.01 #stop:
1.0 #step: 0.01 ) ##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0 0.566667 0
0.0859375 0 0.6 ) #model: #trebleRatio #isReadOnly: true #type: #number ) ##(SliderSpec
#layout: ##(LayoutFrame 0 0.0984375 0 0.566667 0 0.329688 0 0.6 ) #model: #trebleRatio
#orientation: #horizontal #start: 0.01 #stop: 1.2 #step: 0.01 ) ##(LabelSpec #layout:
##(LayoutOrigin 0 0.339063 0 0.516667 ) #isOpaque: true #label: 'IACC' ) ##(LabelSpec
#layout: ##(LayoutOrigin 0 0.339063 0 0.566667 ) #isOpaque: true #label: 'Treble Ratio' )
##(InputFieldSpec #layout: ##(LayoutFrame 0 0.0140625 0 0.216667 0 0.0875 0 0.25 )
#model: #seatingSlope #isReadOnly: true #type: #number ) ##(SliderSpec #layout:
##(LayoutFrame 0 0.0984375 0 0.216667 0 0.329688 0 0.25 ) #model: #seatingSlope

```

```
#orientation: #horizontal #start: 0.0 #stop: 60.0 #step: 0.5 ) #(#LabelSpec #layout:  
#(#LayoutOrigin 0 0.339063 0 0.216667 ) #isOpaque: true #label: 'Seating Slope (deg)' )  
)
```

```

Object subclass: #LightPoint
  instanceVariableNames: 'd lat long '
  classVariableNames: "
  poolDictionaries: "
  category: 'ShadingModel'

```

LightPoint methodsFor: 'accessing'

```

distance
  "returns the distance of the lightpoint from the origin"

  ^d

```

```

latitude
  "returns the latitude of the lightpoint from the origin"

  ^lat

```

```

longitude
  "returns the longitude of the lightpoint from the origin"

  ^long

```

LightPoint methodsFor: 'setting'

```

distance: aDistance
  "sets the distance of the eyepoint from the origin"

  d := aDistance.
  self changed

```

```

distance: aDistance latitude: aLatitude longitude: aLongitude
  "sets the location parameters of the lightpoint with respect to the origin"

  d := aDistance.
  lat := (270 + aLatitude) degreesToRadians.
  long := aLongitude degreesToRadians.
  self changed

```

```

latitude: aLatitude
  "sets the latitude of the eyepoint from the origin"

```

```
lat := aLatitude degreesToRadians.
self changed
```

```
longitude: aLongitude
    "sets the longitude of the eyepoint from the origin"
```

```
long := aLongitude degreesToRadians.
self changed
```

```
"-----"
```

```
LightPoint class
    instanceVariableNames: "
```

LightPoint class methodsFor: 'instance creation'

```
new
    "creates a new instance of an lightpoint"
```

```
^super new
```

```

Array variableSubclass: #PointVector
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'ShadingModel'

```

PointVector methodsFor: 'extraction'

```
extractPointWith: aViewingPlaneDistance
```

"extracts and returns the screen coordinates from a point vector based on a viewing plane distance"

```

      ^((aViewingPlaneDistance*(self x/self z)) @ (aViewingPlaneDistance*(self y/self z)))
scaledBy: 5@5.

```

PointVector methodsFor: 'accessing'

```

x
    "returns the x coordinate of the point vector"

```

```
    ^self at: 1
```

```

y
    "returns the y coordinate of the point vector"

```

```
    ^self at: 2
```

```

z
    "returns the z coordinate of the point vector"

```

```
    ^self at: 3
```

```
"-----"
```

```

PointVector class
  instanceVariableNames: "

```

PointVector class methodsFor: 'instance creation'

```
withX: aNumber1 withY: aNumber2 withZ: aNumber3
```

"creates a point vector with X, Y and Z coordinates"

```
| x |  
x := super new: 4.  
x      at: 1 put: aNumber1;  
        at: 2 put: aNumber2;  
        at: 3 put: aNumber3;  
        at: 4 put: 1.  
^x
```

Object subclass: #Plane

instanceVariableNames: 'id points xNormal yNormal zNormal distance color '

classVariableNames: "

poolDictionaries: "

category: 'ShadingModel'

Plane methodsFor: 'accessing'

color

"returns the color of the plane"

^color

distance

"returns the distance of the normalized plane from the origin"

^distance

id

"returns the ID number of the plane"

^id

points

"returns the collection of points of the plane"

^points

xNormal

"returns the X component of the normal of the plane"

^xNormal

yNormal

"returns the Y component of the normal of the plane"

^yNormal

zNormal

"returns the Z component of the normal of the plane"

^zNormal

Plane methodsFor: 'normalizing'

normalized

"computes the plane equation and sets the X Y & Z components of the normal to the plane, and sets the distance of the normalized plane from the origin"

```
| l m n o p q a b c d x |
l := ((self points at: 3) x) - ((self points at: 1) x).
m := ((self points at: 3) y) - ((self points at: 1) y).
n := ((self points at: 3) z) - ((self points at: 1) z).
o := ((self points at: 2) x) - ((self points at: 1) x).
p := ((self points at: 2) y) - ((self points at: 1) y).
q := ((self points at: 2) z) - ((self points at: 1) z).
a := (m*q) - (n*p).
b := (n*o) - (l*q).
c := (l*p) - (m*o).
d := ((a*((self points at: 1) x)) + (b*((self points at: 1) y)) + (c*((self points at: 1) z)))
```

negated.

```
x := ((a squared + b squared + c squared) sqrt + 0.000001) reciprocal.
self xNormal: (a*x); yNormal: (b*x); zNormal: (c*x); distance: (d*x).
```

Plane methodsFor: 'transformation'

transformUsing: aTransMatrix

"transforms the points of the plane using the transformation matrix aTransMatrix and computes the X Y & Z components of normals, and the distance of the plane from the origin"

```
| x |
x := self points collect: [:each | aTransMatrix multiply4: each].
self points: x.
selfnormalized
```

Plane methodsFor: 'extremes'

maximumZ

"returns the points in the plane in the order of decreasing z values"

```
| x |
x := SortedCollection sortBlock: [:p :q | (p z) >= (q z)].
x addAll: self points.
x
```

```

minimumZ
    "returns the points in the plane in the order of increasing z values"

    | x |
    x := SortedCollection sortBlock: [:p :q | (p z) <= (q z)].
    x addAll: self points.
    ^x

```

Plane methodsFor: 'setting'

```

color: aColor
    "sets the color of the plane"

    color := aColor

distance: aDistance
    "sets the distance of the normalized plane from the origin"

    distance := aDistance

id: anId
    "sets the ID number of the plane"

    id := anId

points: aCollectionOfPoints
    "sets the collection of points of the plane"

    points := aCollectionOfPoints

xNormal: anAngle
    "sets the X component of the normal of the plane"

    xNormal := anAngle

yNormal: anAngle
    "sets the Y component of the normal of the plane"

    yNormal := anAngle

zNormal: anAngle
    "sets the Z component of the normal of the plane"

```

```

        zNormal := anAngle
    "-----"

```

```

Plane class
    instanceVariableNames: "

```

```

Plane class methodsFor: 'instance creation'

```

```

withId: anId withPoints: aCollectionOfPoints
    "creates a plane with an ID number and a collection of points"

```

```

    | x |
    x := super new.
    x id: anId; points: aCollectionOfPoints.
    ^x

```

```

withPoints: aCollectionOfPoints
    "creates a plane with a collection of points"

```

```

    | x |
    x := super new.
    x points: aCollectionOfPoints.
    ^x

```

```

Array variableSubclass: #TransMatrix
instanceVariableNames: "
classVariableNames: "
poolDictionaries: "
category: 'ShadingModel'

```

TransMatrix methodsFor: 'multiplying'

multiply4: aPointVector

"multiplies the receiving translation matrix 4x4 array by the point vector and returns a transformed point vector"

```

| x |
x := Array new: 4.
x at: 1 put: (((self at: 1) at: 1)*(aPointVector x))+(((self at: 1) at:
2)*(aPointVector y))+(((self at: 1) at: 3)*(aPointVector z))+(((self at: 1) at:
4)*(aPointVector at: 4)));
x at: 2 put: (((self at: 2) at: 1)*(aPointVector x))+(((self at: 2) at:
2)*(aPointVector y))+(((self at: 2) at: 3)*(aPointVector z))+(((self at: 2) at:
4)*(aPointVector at: 4)));
x at: 3 put: (((self at: 3) at: 1)*(aPointVector x))+(((self at: 3) at:
2)*(aPointVector y))+(((self at: 3) at: 3)*(aPointVector z))+(((self at: 3) at:
4)*(aPointVector at: 4)));
x at: 4 put: (((self at: 4) at: 1)*(aPointVector x))+(((self at: 4) at:
2)*(aPointVector y))+(((self at: 4) at: 3)*(aPointVector z))+(((self at: 4) at:
4)*(aPointVector at: 4))).
^PointVector withX: (x at: 1) withY: (x at: 2) withZ: (x at: 3)
"-----"

```

```

TransMatrix class
instanceVariableNames: "

```

TransMatrix class methodsFor: 'instance creation'

viewing: anEyePoint

"creates a transformation matrix for viewing from an eyepoint"

```

| x |
x := super new: 4.

```

x at: 1 put: (Array with: (anEyePoint longitude sin negated) with: (anEyePoint longitude cos) with: 0 with: 0);

at: 2 put: (Array with: ((anEyePoint latitude cos)*(anEyePoint longitude cos) negated) with: ((anEyePoint latitude cos)*(anEyePoint longitude sin) negated) with: (anEyePoint latitude sin) with: 0);

at: 3 put: (Array with: ((anEyePoint latitude sin)*(anEyePoint longitude cos) negated) with: ((anEyePoint latitude sin)*(anEyePoint longitude sin) negated) with: (anEyePoint latitude cos negated) with: (anEyePoint distance));

at: 4 put: (Array with: 0 with: 0 with: 0 with: 1).

^x

```

Object subclass: #EyePoint
  instanceVariableNames: 'd lat long '
  classVariableNames: "
  poolDictionaries: "
  category: 'ShadingModel'

```

EyePoint methodsFor: 'accessing'

```

distance
  "returns the distance of the eyepoint from the origin"

```

```

  ^d

```

```

latitude
  "returns the latitude of the eyepoint from the origin"

```

```

  ^lat

```

```

longitude
  "returns the longitude of the eyepoint from the origin"

```

```

  ^long

```

EyePoint methodsFor: 'setting'

```

distance: aDistance
  "sets the distance of the eyepoint from the origin"

```

```

  d := aDistance.
  self changed

```

```

distance: aDistance latitude: aLatitude longitude: aLongitude
  "sets the location parameters of the eyepoint with respect to the origin"

```

```

  d := aDistance.
  lat := (270 + aLatitude) degreesToRadians.
  long := aLongitude degreesToRadians.
  self changed

```

```

latitude: aLatitude
  "sets the latitude of the eyepoint from the origin"

```

```
lat := aLatitude degreesToRadians.
self changed
```

```
longitude: aLongitude
    "sets the longitude of the eyepoint from the origin"
```

```
long := aLongitude degreesToRadians.
self changed
```

```
"-----"
```

```
EyePoint class
    instanceVariableNames: "
```

EyePoint class methodsFor: 'instance creation'

```
new
    "creates a new instance of an eyepoint"
```

```
^super new
```

REFERENCES

- Alexander, C., "The City is not a Tree," *Architectural Forum*, April & May, 1965, pp. 58-62; pp. 58-61.
- Ando, Y., Concert Hall Acoustics, Springer-Verlag, Berlin, 1985.
- Archea, J., "Puzzle-Making: What Architects do when No One is Looking," in Computability of Design, Kalay, Y. E., Ed., John Wiley & Sons, New York, 1987.
- Archer, L. B., Systematic Method for Designers, The Design Council, London, 1965.
- Arnheim, R., Visual Thinking, University of California Press, Berkeley, California, 1969.
- Bakhtin, M. M., The Dialogic Imagination: Four Essays, Holquist, M., Ed., (Translated by C. Emerson and M. Holquist), University of Texas Press, Austin, Texas, 1981.
- Barron, M., "Subjective Study of British Symphony Concert Halls," *Acustica*, Vol. 66, No. 1, June, 1988, pp. 1-14.
- Barron, M., "The Subjective Effects of First Reflections in Concert Halls--The Need for Lateral Reflections," *Journal of Sound and Vibration*, Vol. 15, 1971, pp. 475-494.
- Barron, M., and J. Lee, "Energy Relations in Concert Auditoriums," *Journal of the Acoustical Society of America*, Vol. 84, No. 2, August, 1988, pp. 618-628.
- Barron, M. and A. H. Marshall, "Spatial Impression Due to Early Lateral Reflections in Concert Halls: The Derivation of a Physical Measure," *Journal of Sound and Vibration*, Vol. 77, No. 2, 1981, pp. 211-232.
- Beranek, L. L., Music, Acoustics and Architecture, John Wiley, New York, 1962.
- Blauert, J., and W. Lindemann, "Explorative Studies on Auditory Spaciousness," *Proceedings of the Vancouver Symposium on Acoustics and Theatre Planning for the Performing Arts*, August, 1986, pp. 29-34.

- Borish, J., "Some New Guidelines for Concert Hall Design Based on Spatial Impression," Technical Report, Droid Works, San Raphael, California, Unpublished manuscript.
- Bradley, J. S., "The Evolution of Newer Auditorium Acoustics Measures," *Canadian Acoustics*, Vol. 18, No. 4, 1990, pp. 13-23.
- Bradley, J. S., "Auditorium Acoustics Measures from Pistol Shots," *Journal of the Acoustical Society of America*, Vol. 80, No. 1, July, 1986a, pp. 199-205.
- Bradley, J. S., "Predictors of Speech Intelligibility in Rooms," *Journal of the Acoustical Society of America*, Vol. 80, 1986b, pp. 837-845.
- Bradley J. S., "Speech Intelligibility Studies in Classrooms," *Journal of the Acoustical Society of America*, Vol. 80, 1986c, pp. 846-854.
- Bradley, J. S. and R. E. Halliwell, "Making Auditorium Acoustics More Quantitative," *Sound and Vibration*, February, 1989, pp. 16-23.
- Broadbent, G., Design in Architecture, John Wiley & Sons, Chichester, England, 1973.
- Byte, (Special issue on Smalltalk), August, 1981.
- Chiang, Wei-Hwa, "Effects of Various Architectural Parameters on Six Room Acoustical Measures in Auditoria," Ph.D. Dissertation, University of Florida, Gainesville, 1994.
- Cremer, L., Principles and Applications of Room Acoustics, Vol. 1, (Translated by T. Schultz), Applied Science Publishers, London, England, 1978.
- Cross, N. C., The Automated Architect, Pion Limited, London, 1977.
- de Champeaux, D., and W. Olthoff, "Towards an Object-oriented Analysis Technique," *Proceedings of the Pacific Northwest Software Quality Conference*, Portland, Oregon, September, 1989, pp. 323-338.
- Doelle, L. L., Environmental Acoustics, McGraw-Hill, New York, 1972.
- Dijkstra, E. W., "Notes on Structured Programming," in Structured Programming, Dahl, O. J., E. W. Dijkstra and C. A. R. Hoare, Eds., Academic Press, London, 1972.
- Eastman, C. M., "The Evolution of CAD: Integrating Multiple Representations," *Building and Environment*, Vol. 26, No. 1, 1991, pp. 17-23.

- Eastman, C. M., "Fundamental Problems in the Development of Computer-Based Architectural Design Models, in Computability of Design, Kalay, Y. E., Ed., John Wiley & Sons, 1987.
- Eastman, C. M., "Abstractions: A Conceptual Approach for Structuring Interaction with Integrated CAD Systems," *Computers and Graphics*, Vol. 9, No. 2, 1985, pp. 97-105.
- Evans, D. C., "Computer Logic and Memory," *Scientific American*, September, 1966, pp. 74-85.
- Eysholdt, U., D. Gottlob, K. F. Siebrasse and M. R. Schroeder, "Raumlichkeit und Halligkeit - Untersuchung zur Auffindung korrespondierender objectiver Parameter," *DAGA-75 (Deutsche Gesellschaft Fur Akustik)*, 1975, p. 471.
- Gade, A. C., "Acoustical Survey of Eleven European Concert Halls," Report No. 44, The Acoustics Laboratory, Technical University of Denmark, Lyngby, 1989.
- Gade, A. C., "Relationships Between Objective Room Acoustic Parameters And Concert Hall Design," *Proceedings of the 12th International Congress on Acoustics*, Vol./Band D-G, E4-8, Toronto, 1986.
- Glassner, A., An Introduction to Ray Tracing, Academic Press, London, England, 1989.
- Goldberg, A., and D. Robson, Smalltalk-80: The Language, Addison-Wesley, Menlo Park, California, 1989.
- Guesgen, H. W., and J. Hertzberg, A Perspective of Constraint-Based Reasoning, Springer-Verlag, Berlin, 1992.
- Haas, H., "The Influence of a Single Echo on the Audibility of Speech," *Journal of the Auditory Engineering Society*, Vol. 20, 1972, pp. 146-159.
- Harfmann, A. C., and S. S. Chen, "Building Representation within a Component Based Paradigm, *Proceedings of the ACADIA Conference*, Big Sky, Montana, 1990, pp. 117-127.
- Hawkes, R. J., and H. Douglas, "Experience in Concert Auditoria," *Acustica*, Vol. 24, 1971, pp. 236-250.
- Hook, J. L., "Acoustical Variation in the Foellinger Great Hall, Krannert Center for the Performing Arts," Master's Thesis, University of Illinois, Urbana, 1989.
- Houtgast, T., and H. J. M. Steeneken, "The Modulation Transfer Function in Room Acoustics as a Predictor of Speech Intelligibility," *Acustica*, Vol. 28, 1973, pp. 66-73.

- Houtgast, T., H. J. M. Steeneken and R. Plomp, "A Physical Method for Measuring Speech Transmission Quality," *Journal of the Acoustical Society of America*, Vol. 46, 1980, pp. 60-72.
- Izenour, G. C., Theater Design, McGraw-Hill, New York, 1977.
- Jones, J. C., "Design Methods Reviewed," in The Design Method, Gregory, S., Ed., Butterworth, London, 1966, pp. 295-309.
- Jullien, J. P., "Correlations Among Objective Criteria of Room Acoustic Quality," *Proceedings of the 12th International Congress on Acoustics*, Vol./Band D-G, E4-9, Toronto, 1986.
- Kalay, Y. E., Ed., Evaluating and Predicting Design Performance, John Wiley & Sons, New York, New York, 1992.
- Kalay, Y. E., Modeling Objects and Environments, John Wiley & Sons, New York, New York, 1989.
- Kalay, Y. E., Ed., Computability of Design, John Wiley & Sons, New York, New York, 1987a.
- Kalay, Y. E., "Worldview: An Integrated Geometric-Modeling/Drafting System," *IEEE Computer Graphics & Applications*, February, 1987b, pp. 36-46.
- Kay, A. C., "Microelectronics and the Personal Computer," *Scientific American*, September, 1977, pp. 230-244.
- Kay, A. C., and A. Goldberg, "Personal Dynamic Media," *Computer*, Vol. 10, 1977, pp. 31-41.
- Korson, T., and J. D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol. 33, No. 9, September, 1990, pp. 40-60.
- Krasner, G. E., Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, Menlo Park, California, 1983.
- Krasner, G. E., and S. T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, August/September, 1988, pp. 26-29.
- Kuhn, T., The Structure of Scientific Revolutions, University of Chicago Press, Chicago, 1962.

- LaLonde, W. R., and J. R. Pugh, Inside Smalltalk Vols. 1 & 2, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
- Ledbetter, L. and B. Cox, "Software-ICs," in *Byte*, June, 1985, pp. 307-316.
- Lehman, P., and H. Wilkens, "Zusammenhang subjektiver Beurteilung von Konzertsälen mit raumakustischen Kriterien," *Acustica*, Vol. 45, 1980, pp. 256-268.
- Lochner J. P. A., and J. F. Burger, "The Influence of Reflections on Auditorium Acoustics," *Journal of Sound Vibration*, Vol. 1, 1964, pp. 426-454.
- Lochner J. P. A., and J. F. Burger, "The Subjective Masking of Short Time Delayed Echoes by Their Primary Sounds and Their Contribution to the Intelligibility of Speech," *Acustica*, Vol. 8, 1958, pp. 1-10.
- Luckmann, J., "An Approach to the Management of Design," in Developments in Design Methodology, Cross, N., Ed., John Wiley & Sons, Chichester, England, 1984, pp. 83-97.
- Maxfield, J. P., and W. J. Albersheim, "An Acoustic Constant of Enclosed Spaces Correlateable with Their Apparent Liveness," *Journal of the Acoustical Society of America*, Vol. 19, 1947, pp. 71-79.
- McKim, R. H., Experiences in Visual Thinking, Second Edition, PWS Publishers, Boston, Massachusetts, 1980.
- Mitchell, W. J., "A New Agenda for Computer-aided Architectural Design," *Proceedings of the ACADIA Conference*, Gainesville, Florida, 1989, pp. 27-43.
- Mitchell, W. J., Computer-Aided Architectural Design, Van-Nostrand Reinhold, New York, 1977.
- Mitchell, W. J., M. McCullough and P. Purcell, Editors, The Electronic Design Studio: Architectural Knowledge and Media in the Computer Era, MIT Press, Cambridge, Massachusetts, 1990.
- Newell, A., and H. A. Simon, Human Problem Solving, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- Papanek, V., Design for the Real World, Thames and Hudson, London, 1972.
- Reichardt, W., and U. Lehman, "Optimierung von Raumeindruck und Durchsichtigkeit von Musikdarbietungen durch Auswertung von Impulsschalltests," *Acustica*, Vol. 48, 1981, pp. 174-185.

- Reisig, W., A Primer in Petri Net Design, Springer-Verlag, Berlin, 1992.
- Rowe, P. G., Design Thinking, The MIT Press, Cambridge, Massachusetts, 1987.
- Sabine, W. C., Collected Papers on Acoustics, Harvard University Press, Cambridge, Massachusetts, Reprinted by Dover Publications, 1964.
- Schroeder, M. R., "Modulation Transfer Function: Definition and Measurement," *Acustica*, Vol. 49, 1981, pp. 179-182.
- Schroeder, M. R., B. S. Atal and G. M. Sessler, "Subjective Reverberation Time and its Relation to Sound Decay," *Proceedings of the 5th ICA*, Paper G32, Leige, 1965.
- Schultz, T. J., "Acoustics of the Concert Hall," *IEEE Spectrum*, Vol. 2, June, 1965, pp. 56-67.
- Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Ada Letters*, Vol. 7, July/August, 1987, pp. 54-67.
- Shu, N.C., Visual Programming, Van Nostrand Reinhold Company, New York, 1988.
- Siebein, G. W., Acoustical Modeling Workshop, Course notes for ARC (Architecture) 7796, University of Florida, Gainesville, 1989.
- Simon, H. A., "The Structure of Ill-structured Problems," in Developments in Design Methodology, Cross, N., Ed., John Wiley & Sons, Chichester, England, 1984.
- Simon, H. A., The Sciences of the Artificial, MIT Press, Cambridge, Massachusetts, 1969.
- Smith, D. N., Concepts of Object-Oriented Programming, McGraw Hill, New York, 1991.
- Stettner, A., "Computer Graphics for Acoustic Simulation and Visualization," Master's Thesis, Cornell University, Ithaca, New York, 1989.
- Tan, M., "Closing in on an Open Problem--Reasons and a Strategy to Encode Emergent Subshapes," *Proceedings of the ACADIA Conference*, Big Sky, Montana, 1990, pp. 5-19.
- Thiele, R., "Richtungsverteilung und Zeitfolge der Schallruckwürfe in Raumen," *Acustica*, Vol. 3, 1953, pp. 291-302.
- Thorndike, E. L., Human Learning, MIT Press, Cambridge, Massachusetts, 1931.
- Wegner, P., "Learning the Language," *Byte*, March, 1989, pp. 245-253.

Wirfs-Brock, R. J., and R. E. Johnson, "Surveying Current Research in Object-oriented Design," *Communications of the ACM*, September, 1990, pp. 104-124.

Yessios, C. I., "The Computability of Void Architectural Modeling," in Computability of Design, Kalay, Y. E., Ed., John Wiley & Sons, New York, 1987.

Yessios, C. I., "What has yet to be CAD," *ACADIA Workshop '86 Proceedings*, Houston, October, 1986, pp. 29-36.

BIOGRAPHICAL SKETCH

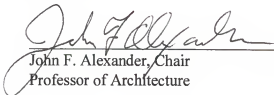
Ganapathy Mahalingam was born on April 23, 1961, in Madras, India. He was the eldest of three children of Mr. Mahalingam, an abrasives specialist, and Lakshmi Mahalingam. He attended the Don Bosco Matriculation Higher Secondary School, Madras, from which he graduated in 1977.

He pursued an undergraduate education in architecture at the School of Architecture and Planning, Madras, from 1978 to 1983. He received the professional degree of Bachelor of Architecture from the University of Madras in 1984 after completing a year of practical training at Kharche and Associates, Madras. He continued to work for another year at Kharche and Associates as an architect. He designed many residential buildings, commercial buildings and educational institutions during his tenure at Kharche and Associates.

He left India for the United States of America in August, 1985, to pursue a master's degree in architecture at Iowa State University. During his graduate education, he specialized in the field of computer-aided design in architecture. He obtained the post-professional Master of Architecture degree from Iowa State University in August, 1986. He then taught for a year in Iowa State University's Department of Architecture as an instructor from August, 1986, to August, 1987. His teaching responsibilities included all of the department's computer-related courses. He returned to India in August, 1987, and engaged in private practice until 1989. He designed and executed interiors and a small residential building.

He left India once again for the United States of America in August, 1989, to pursue a Ph.D. degree in architecture at the University of Florida, which was awarded in May, 1995.

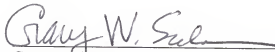
I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


John F. Alexander, Chair
Professor of Architecture

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


Earl M. Starnes, Cochair
Professor Emeritus of Urban and Regional
Planning

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


Gary W. Siebein
Professor of Architecture

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


Anthony J. Dasta
Professor of Architecture

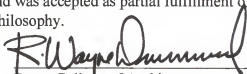
I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Upendranatha S. Chakravarty
Associate Professor of Computer and
Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Architecture and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May, 1995



Dean, College of Architecture

Dean, Graduate School